R Programming: Zero to Pro

Yang Feng and Jianan Zhu

2021-08-24

Contents

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/#downloa
d

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.

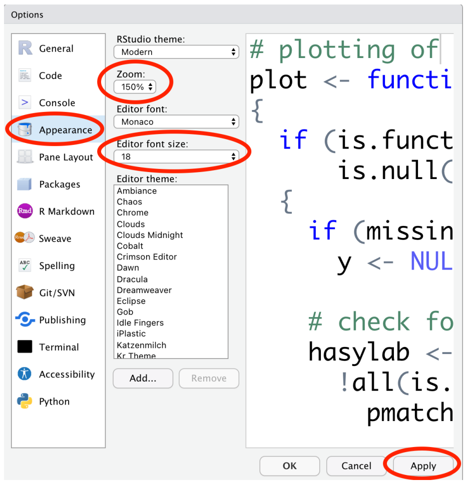
a. Customize appearance

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.



RStudio theme: Modern
Zoom: 150%
Editor font: Monaco
Editor font size: 18

```
# plotting of
plot <- functi
{
  if (is.funct
      is.null(
  {
    if (missin
      y <- NUL

    # check fo
    hasylab <-
      !all(is.
        pmatch
```

Figure 1.1: Zoom and Editor font size

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

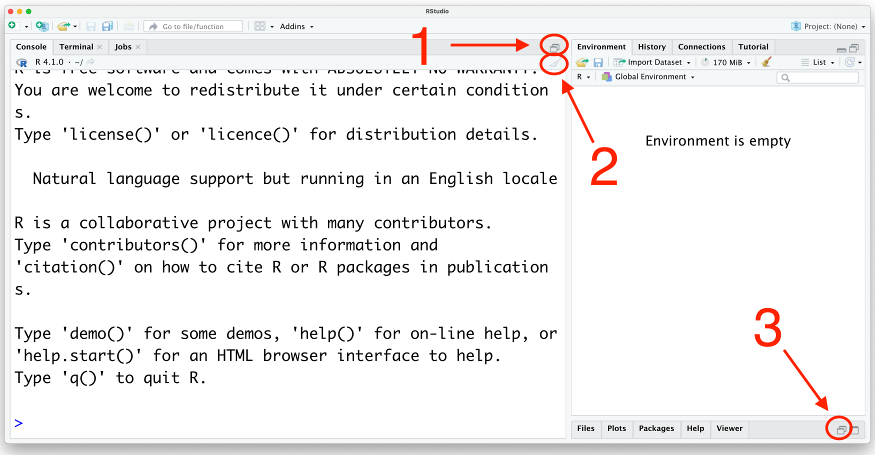
b. Four panels of RStudio

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. [1] By clicking button 2, we can clear the content in the bottom left panel as shown in the following figure.

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.

[1]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.



```
You are welcome to redistribute it under certain condition
s.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publication
s.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure 1.2: Unfold panels

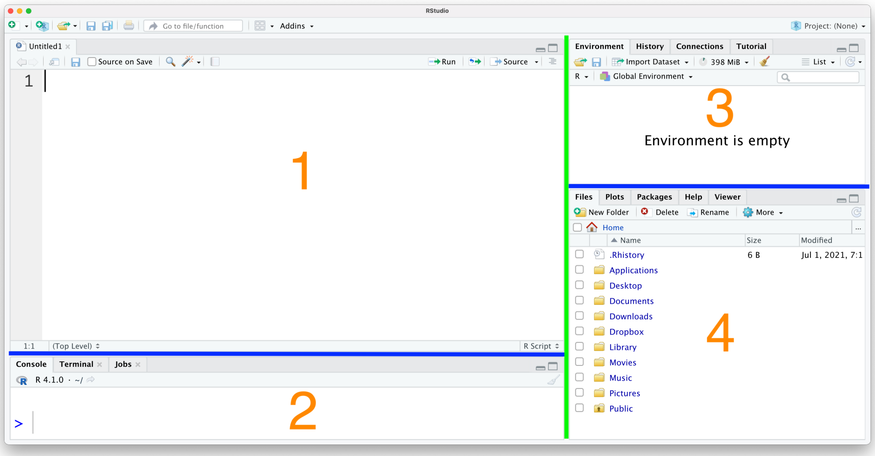


Figure 1.3: Four panels

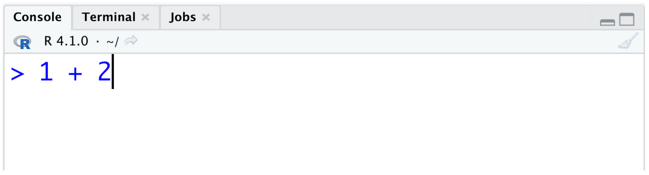
- Panels 1 and 2 are located to the left of the green line, and are collectively called the **Code Area**. We will introduce them next.
- Panels 3 and 4 are located to the right of the green line, and are collectively called **R Support Area**. We will introduce these two panels in later sections. **Add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

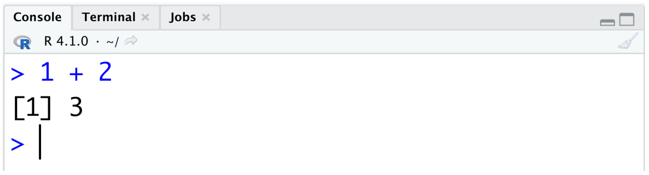
It is a good habit to add spaces around an operator to increase readability of the code.



```
> 1 + 2
```

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.



```
> 1 + 2
[1] 3
> 
```

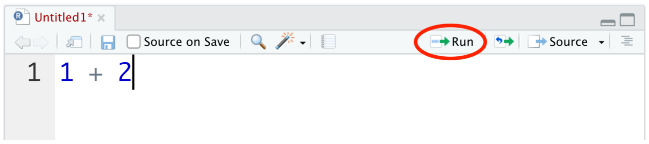
Figure 1.5: R code(2)

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 Cmd+Return on Mac or Ctrl+Enter on Windows.

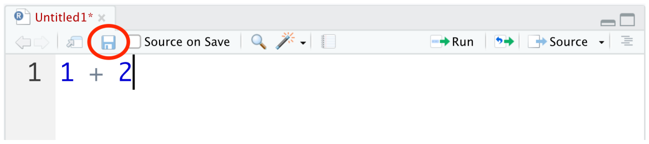


```
1 + 2
```

Figure 1.6: script

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

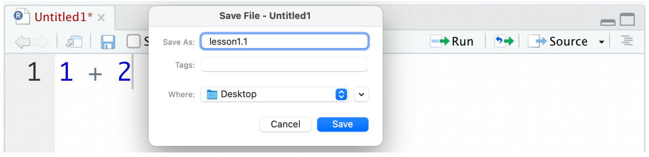
After finishing writing codes in the editor, you can save them as a script. To do that, you can click the *Save* button as shown in the Figure 1.7. The keyboard shortcut of saving files is Cmd+S on Mac or Ctrl+S on Windows.



```
1 + 2
```

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.



```
1 + 2
```

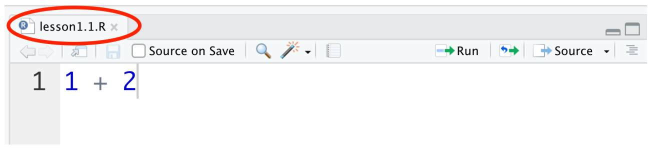
Save As: lesson1.1

Where: Desktop

Figure 1.8: Save (II)

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

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.



```
1 + 2
```

Figure 1.9: Save (III)

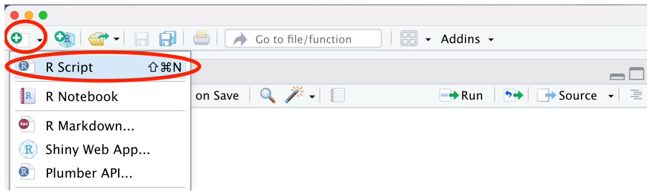


Figure 1.10: create a new script

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.

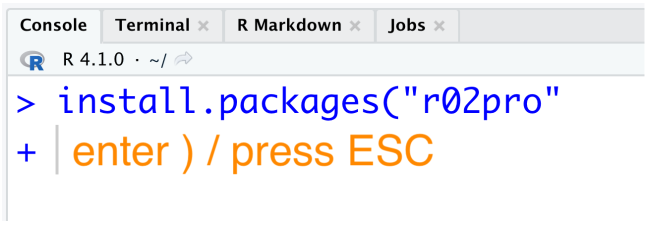
To install an R package, you need to use a built-in R **function** , which is `install.packages()`. A **function** takes in **arguments** (inputs) and performs a specific task. After the function name, we always need to put **a pair of parentheses** with the arguments inside.

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 introduce 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.



```
> install.packages("r02pro"
+ enter ) / press ESC
```

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

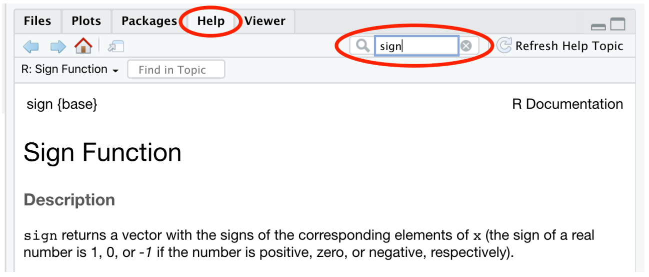
Now let's start to use R as a calculator! You can use R to do addition, subtraction, multiplication division, and combine multiple basic operations. You can also calculate the square root, absolute value and the sign of a number.

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



sign {base} R Documentation

Sign Function

Description

`sign` returns a vector with the signs of the corresponding elements of `x` (the sign of a real number is 1, 0, or -1 if the number is positive, zero, or negative, respectively).

Figure 1.12: Ask for help

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

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

We all know that $7 = 3 * \mathbf{2} + \mathbf{1}$. 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 $\mathbf{2} <= 7/3 <= \mathbf{3}$, you can use the `floor` function to find the *largest integer* $<= 7/3$, which is 2; and the `ceiling` function gives the *smallest integer* $>= 7/3$, which is 3.

c. Round to the nearest number

```r
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,

$$1 \times 10^5 = \text{1e+05} \tag{1.1}$$
$$2 \times 10^4 = \text{2e+04} \tag{1.2}$$
$$1.2 \times 10^{-3} = \text{1.2e-03} \tag{1.3}$$

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*.

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

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

```r
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 $acos(b) = a$.

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

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

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

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

1.2.6 Exercise

You can run the following code to do the exercise.

```
r02pro(1.2)
```

Chapter 2

R Objects

In 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 `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 name instead.

2.1 Object Assignment

2.1.1 Object

Firstly, we will introduce the most important thing in R, which is called **object**. In principle, **everything that exists in R is an object**. For example, the number `5` is an object, the expression `1 + 2` is an object, and `floor(7 / 3)` is also an object.

In the above examples, there is only one **element** in the object. However, the object can contain more than one elements, and each element has its own **value**. Notice that different elements can have the same value. Values can be of three types including numerical, character, and logical.

For example, `1 + 2` is an object with one element of value 3.

2.1.2 Assignment Operation with `<-`

Knowing the importance of objects, let's introduce how to do **object assignments** in R. To do object assignments, you assign **value(s)** to a **name** via the assignment operator, which will create a new object with a name. You can use the new named object once it is created in subsequent calculations without redundancy. Let's start with a simple example,

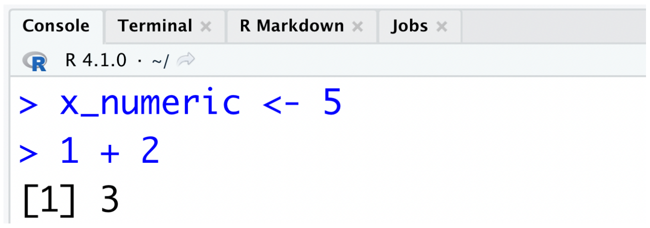
```
x_numeric <- 5
```

The assignment operation has three components. From left to right

- the first component `x_numeric` is the **object name** of a new object, which has certain naming rules which we will discuss shortly in Section 2.1.4.
- 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(s)** 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?



```
> 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. (For named objects, you can get their values by running codes with their object names.)

```r
x_numeric
```

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

Great! You get the value 5, indicating that you have successfully assigned the value 5 to the name x_numeric, and you have created a new object `x_numeric`. You can use `x_numeric` instead of `5` to do the subsequent calculations because `x_numeric` and `5` have the same value.

You can also assign value(s) of any R expression (including operations and functions) to a name. Sometimes, the value(s) of an R expression may be unintuitive. To get the value(s) of an R expression, R will get the result of it firstly.

In this case, R will first calculate the result of `exp(3) / log(20,3) * 7` and assign the value of the result to a name. Let's see the following example.

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

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

Now you have successfully created an object `y_numeric` with value 51.56119. Using the named object `y_numeric`, you can do the same three calculations introduced at the beginning of this chapter as follows.

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

You can also try the following examples by yourself.

```r
a <- floor(7 / 3)
a
b <- 7%/%3
b
```

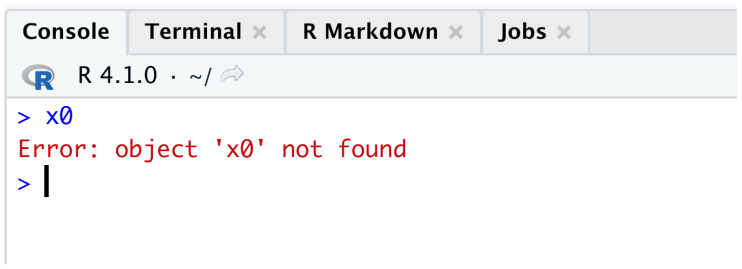
Clearly, using the object assignment, we can greatly simplify our code and avoid redundancy.

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

```r
X_numeric
```

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

You can only use the names of named objects to get their values. If you type any unassigned name, e.g. x0, you will see an error.



```
> x0
Error: object 'x0' not found
>
```

Figure 2.2: x0 is not an object

2.1.3 Review objects in environment

After creating new objects `x_numeric` and `y_numeric`, they will appear in the **Environment**, located in the top right 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 look fine. Notice that objects without names will not be shown in the environment.

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

```
ls()
```

```
#>  [1] "a"                   "all_letters"         "ani_char"
#>  [4] "animal"              "animal_tidy"         "animal_wide"
#>  [7] "animal_wide_weight"  "b"                   "char_vec"
#> [10] "Code"                "comp_seq1"           "comp_seq3"
#> [13] "d"                   "dig_num"             "dig_num_new"
#> [16] "Explanation"         "extend"              "g"
#> [19] "g1"                  "lower_letters"       "my_list"
#> [22] "n"                   "Name"                "norm_dat"
#> [25] "norm_dat_1"          "norm_dat_2"          "norm_dat_3"
#> [28] "num1"                "num2"                "num3"
#> [31] "nums"                "Operation"           "p"
#> [34] "para_1"              "para_2"              "Pattern"
#> [37] "pattern1"            "pattern2"            "pattern3"
#> [40] "pattern4"            "q"                   "Section"
#> [43] "syms"                "upper_letters"       "weight"
#> [46] "x"                   "x_numeric"           "x1"
#> [49] "x2"                  "x3"                  "y_numeric"
#> [52] "year"
```

All the objects shown in the environment or the list have been saved in R, so they are available for future use directly. It is a good habit to do object assignments if you want to save important values for future use.

2.1.4 Object naming rule

Now you have created two named objects `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 lowercase 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

2.1.5 Object types

In this section, you have learned about how to assign a value to a name. The values you assigned are all of numeric type. Actually, an object may contain more than one values. Also, the values it contains can be of other types than numeric, including character and logical. Depending on the **composition of values**, 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.

2.1.6 Exercise

You can run the following code to do the exercise.

```
r02pro(2)
```

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 type 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 values of numeric type. For example, `6` is a numeric vector with one element of value 6. For vectors, the number of elements corresponds to the length of vector, so `6` is a numeric vector with length 1.

After assigning the value 6 to the name x1, you have created a new vector `x1` with the same value as `6, soxlis also a numeric vector. And you can refer tox1`‘ in the subsequent calculations.

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

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

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

In this example, you have created a length-5 object using the `c()` function with arguments containing the five elements separated by comma. Since the value of each element is a number, the object is a numeric vector.

If you assign the values to the name y1, you will get a new numeric vector `y1` with 5 values. Notice that the second and third elements have the same value 3 in `y1`. You can verify the contents of `y1` and check the length of it through the `length()` function.

When you assign several values to a name, the order of the values will not change after assignment. If you create two numeric vectors with same numbers of different orders, these objects will have different values. For example,

```
y2 <-  (1, 3, 5, 7, 9)
y2
y3 <-  (9, 7, 5, 3, 1)
y3
```

Here, `y2` and `y3` have different values.

If you include several numeric vectors in `c()`, you will also create a numeric vector as a combination of the input numeric vectors. For example, you can create a numeric vector with values from two numeric vectors. Of course you can create a new numeric vector `z1` using object assignment.

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

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

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

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

b. Operations between two numeric 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 perform the operation separately for each element. First, let's create another vector `x2` 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 value of the first element is $1 + 2 = 3$, and value of the second element is $3 + 4 = 7$, etc.

Since the calculation is done elementwisely, people 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 followings are a few additional examples you can try.

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

2.2.2 Character vector

a. Create character vectors

Now, let's move to character vectors. In a **character vector**, the value of each element is of character type, which means each value is a **string**. A **string** is a sequence of characters (including 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 `sheepstudio` which only has one element. 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.

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. ("4" and "29" are strings)

```
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 elements,

```
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
```

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

```
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 in `paste()`, the default separator between the individual strings is space. Actually you can change the separator by setting the `sep` argument in `paste()`. For example, you can separate the individual 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. Let's review the character vector `four_strings` at first,

```
four_strings <- c("This", "is", "Sheep@29", "$Studio")
four_strings
```

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

Then use the `tolower()` function to convert all letters to lower case,

```
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"
```

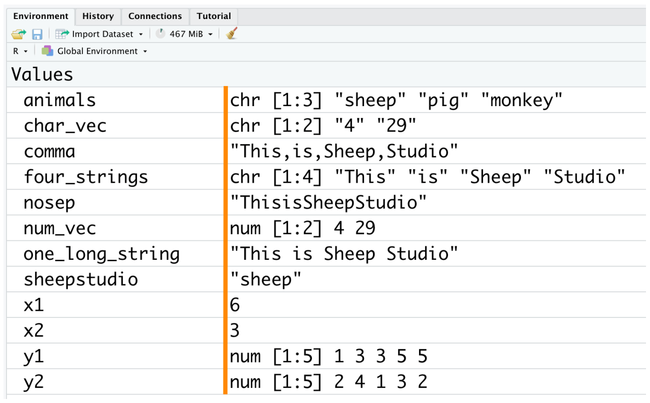
2.2.3 Logical vector

So far we have created several numeric vectors and character vectors. Some vectors have names, and some are not. You can see all the **named objects** by using the `ls()` function.

```
ls()
```

```
#>  [1] "a"                   "all_letters"        "ani_char"
#>  [4] "animal"              "animal_tidy"        "animal_wide"
#>  [7] "animal_wide_weight" "animals"            "b"
#> [10] "char_vec"            "Code"               "comma"
#> [13] "comp_seq1"           "comp_seq3"          "d"
#> [16] "dig_num"             "dig_num_new"        "Explanation"
#> [19] "extend"              "four_strings"       "g"
#> [22] "g1"                  "key_mat"            "Keys"
#> [25] "lower_letters"       "my_list"            "n"
#> [28] "Name"                "norm_dat"           "norm_dat_1"
#> [31] "norm_dat_2"          "norm_dat_3"         "nosep"
#> [34] "num_vec"             "num1"               "num2"
#> [37] "num3"                "nums"               "one_long_string"
#> [40] "Operation"           "p"                  "para_1"
#> [43] "para_2"              "Pattern"            "pattern1"
#> [46] "pattern2"            "pattern3"           "pattern4"
#> [49] "q"                    "Section"              "sheepstudio"
#> [52] "syms"                 "Type"                 "upper_letters"
#> [55] "weight"               "x"                    "x_numeric"
#> [58] "x1"                   "x2"                   "x3"
#> [61] "y_numeric"            "y1"                   "y2"
#> [64] "y3"                   "year"                 "z1"
```

As introduced in Section 2.1, another way to check the named objects is via the environment panel as shown in Figure 2.4.



Values

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 2.4: Environment

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.

By now you have created several objects, and you will find that the objects will not be saved in R if you don't assign their values to names, for example, the results of `x1 + x2` and `y1 + y2` are not shown in the environment.

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

```r
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 vector is of logical type, or you can use `class()` to find out the exact type.

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

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

```r
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.

```r
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.

```r
tlogic <- True
```

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

2.2.4 The coercion rule

So far, you have known that vectors are objects that have values of the same type, including numbers, strings, or logical values. But in practice, you may have values with a mix of different types. If you still want to combine them into 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).

$$logical < numeric < character$$

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

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

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

```
class(mix_1)
```

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

You can see that `TRUE` will be converted to 1 and `FALSE` will be converted to 0 when they appear with numbers, that's because numbers are more complex than logical values, and R will unify all values into the most complex one. Then you will see that `mix_1` is a numeric vector with four numbers. This is the **most commonly** usage of coercion rule in R.

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"
```

You can see that both `8` and `26` are converted into strings since strings are more complex than numbers. Then `mix_2` will be a character vector.

The final example mixes logical values, numbers and strings.

```
mix_3 <- c(16, TRUE, "pig")
mix_3
```

```
#> [1] "16"   "TRUE" "pig"
```

```
class(mix_3)
```

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

You can see in `mix_3`, both `97` and `TRUE` are converted to strings! That's because values of character type are the most complex among all values.

```
mix_4 <- c(c(16, TRUE), "pig")
mix_4
```

```
#> [1] "16"  "1"   "pig"
```

However, if you create another vector `mix_4`, you first have `c(97, TRUE)` which will be converted to `c(97, 1)` since numbers are more complex than logical values. Then, `c(97, 1)` will be converted to `c("97", "1")` when you combine it with `"pig"`, leading to the results of `mix_4`.

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 values of **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()` together 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 equally-spaced 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 5` as its values.

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

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

Note that the end value of the sequence doesn't always equal the `to` argument. If you 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. This is the reason why `to` is called the limited end value, not the end value.

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

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

Unlike `:`, you can set values of three arguments in `seq()` as decimal numbers.

```
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
```

Here, you will get a sequence which starts with 1.1, increases by 0.7 each time until it is larger than 6.2.

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 equally-spaced 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.

```r
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.

```r
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.

```r
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 matching numeric vectors via `seq_along()`

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

```r
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()`.

```r
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, you 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 composed 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 of value 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, the `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(4,2), 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 (elements of different values) 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 `1`s, two `5`s 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 part.

```
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)
```

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**.

```r
rank(x)
```

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

If you check the values 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.

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

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

If you want to break the ties by the order element appears in the vector, you can set `ties.method = "first"`. Then the earlier appearing element will have smaller ranks than the later one. 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 documentation of `rank()` if interested.

```r
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, `order()` is a very *different* function from `sort()`.

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

```r
x
```

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

```r
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.

```r
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.

```r
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 appear first, followed by digits, and letters come last.
- symbols are ordered in the following way.

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

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

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

```r
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.

```r
all_letters <- c(letters,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"
```

Here, `letters` is a character vector precreated by R, it has all 26 letters in the alphabet with lower case. And `LETTERS` is another character vector, which has all 26 letters in the alphabet with upper case.

b. Sort vectors

As before, you can apply `sort()` on character vectors. Basically, the elements of character vectors ordered by the first character of their values, move 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.

```r
sort(char_vec)
```

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

We have the following observations.

- Symbols appear first, followed by digits, and letters come last.
- 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 rank 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 indices for each element in character vectors with the same `order()` function like that for numeric vectors. Also, the `order()` function breaks the ties by the appearing order by default.

```
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)
```

```r
rank(logi_vec)
order(logi_vec)
```

2.4.4 Exercise

You can run the following code to do the exercise.

```r
r02pro(2)
```

2.5 Statistical Functions on Vectors

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

2.5.1 Numeric vectors

Let's first create a numeric vector.

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

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

Group A: minimum and maximum

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

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

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

In addition to getting the minimum and the maximum values, it is often useful to get the corresponding locations of them. Here, the fourth element in `h` has the minimum value 0, so you will get a result of 4 from `which.min()`. If there are multiple elements with the minimum value, `which.min()` will return the first location. Similarly, `which.max()` tells you the location of the maximum value.

```
g <-  (2, 2, 1, 1)
         (g)
         (g)
```

The third element and the fourth element in `g` both have the minimum value, but `which.min(g)` has a value of 3 since the third element is the first location with the minimum value. Similarly, `which.max()` gives you a result of `1`.

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

In addition to calculating the minimum value of all elements, you can also use 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 of `cummin(h)` is `3` since the minimum of the first element in the original vector is always itself, the second element is `2` since the minimum of the first two elements (`3` and `2`) in `h`is 2, and so on. Note that once we reach the minimum value of the vector, the remaining elements of the cumulative minimum function will always equal to the minimum value. There is also a corresponding function for computing the cumulative maximums, called `cummax()`.

Group B: sum and product

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

Next, let's look at the `sum()` function, which produces the **sum** of all elements in the vector. For the numeric vector `h`, the sum is `3+2+75+0+100`, which is `180`. Similar to `cummin()`, you can use the `cumsum()` function to compute the *cumulative sums*, which works by summing up the elements of the original vector cumulatively up to each location. In `cumsum(h)`, 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`) in `h` is 5, and you can easily verify the value of the remaining elements by yourself.

```
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 is `0`. Again, we have the *cumulative product* function `cumprod()` working by multiplying the elements of the original vector cumulatively up to each location.

Group C: mean and median

```
sort(h)
```

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

Before introducing this group, let's first review the `sort()` function introduced in Section 2.4. By default, this function can sort elements from the smallest to the largest.

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

The `mean()` function returns the average of all elements. And the `median()` function returns the middle number in the resulting vector of `sort()` where the elements are listed in order from the smallest to the largest. If the vector length is odd, the middle number is the value of the element in the central location. In `sort(h)`, we can see that the median corresponds to the third number out of five numbers since there are two numbers larger than 3 and two numbers smaller than 3. If the vector length is even, the middle number is the average of the two middle elements after sorting.

```
    (g)
      (g)
```

Take `g` for example, after sorting, you will see that `1` and `2` are in the middle. The median is then defined as the average of these two elements, equaling 1.5.

Group D: quantiles

```
quantile(h)
```

```
#>   0%  25%  50%  75% 100% 
#>    0    2    3   75  100
```

`quantile()` produces **sample quantiles** of a given numeric vector. By default, it generates 5 numbers, the top row represents the different percentiles, including the 0 percentile, 25th percentile (0.25 quantile), 50th percentile (0.5 quantile), 75th percentile (0.75 quantile), and 100th percentile, and the second row consists of the corresponding values of each quantile. We next go over all five quantile values.

First of all, 0 percentile and 100th percentile are always the minimum and the maximum values, respectively. The 50th percentile (0.5 quantile) is the same as the median.

The 25-th percentile (0.25 quantile), also called the **first quartile**, is the value such that there are 25 percent (or a quarter) of the remaining data (whole data without this number) smaller than it. For vector `h`, the value is 2 since there is exactly 1 number, which is 25 percent of the remaining 4 numbers, smaller than 2.

Similarly, the 75-th percentile, also called the **third quartile**, is the value such that 75 percent of the remaining data is smaller than this number. For vector `h`, the value is 75 since there are 3 numbers, which are 75 percent of the remaining 4 numbers, smaller than 75.

You also have an important concept called **interquartile range** (IQR), defined as the difference between the 3rd quartile (75-th percentile) and the 1st quartile (25-th percentile). The interquartile range of `h` is 73, which is `75 - 2`.

```
IQR(h)
```

In addition to the default five percentiles, you can also use the `quantile()` function to get any quantile between 0 and 1. To do this, you just need to specify the second argument `probs`. Let's try to find the 95th quantile.

```
quantile(h, probs = 0.95)
```

As before, this asks you to compute the 95th percentile, meaning 95 percent of the remaining data is smaller than this value. Because you only have 5 values in this vector, it may not be very intuitive. However, if you have more elements in a vector, say 1001, you can count the number of the remaining data that is smaller than this value, which should be 950 (the number of remaining data is 1000, and 95 percent of 1000 is 950).

In addition, the second argument can be a vector of probabilities, which will produce a numeric vector of the corresponding quantiles.

```
quantile(h, probs = c(0.1, 0.2, 0.99))
```

Group E: summary statistics

```
summary(h)
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>       0       2       3      36      75     100
```

Compared with `quantile()`, a more general function to have a comprehensive understanding of numeric vectors is `summary()`. From `summary()`, you can get the 5 percentiles and the mean.
(Min: 0 percentile, 1st Qu: 25-th percentile, Median: 50-th percentile, 3rd Qu: 75-th percentile, Max: 100-th percentile)

Group F: variance and standard deviation

```
var(h)
sd(h)
```

The last group of functions are `var()` and `sd()` which compute the **sample variance** and **sample standard deviation** of a numeric vector, respectively. The formula of sample variance of vector h is

$$var(h) = \frac{1}{n-1}\sum_{i=1}^{n}(h_i - \bar{h})^2,$$

where n is the length of h and $\bar{h}$ is the average of all elements. By definition, the sample standard deviation is the square root of sample variance, which you can verify by `sqrt(var(h))`.

For your convenience, we would like to provide a summary of all the functions introduced in the following table.

Operation	Explanation
min(h)	the minimum value
max(h)	the maximum value
range(h)	both the minimum value and the maximum value
which.min(h)	the (first) location of the minimum value
which.max(h)	the (first) location of the maximum value
cummin(h)	the cumulative minimum values
cummax(h)	the cumulative maximum values
sum(h)	the sum of all elements
cumsum(h)	the cumulative sum
prod(h)	the product of all elements
cumprod(h)	the cumulative products
mean(h)	the average of all elements
median(h)	the middle number in sort(h)
quantile(h)	the 0 percentile, 25-th percentile, 50-th percentile, 75-th percentile, and 100-t
IQR(h)	the difference between the 3rd quartile and the 1st quartile
quantile(h, probs = 0.95)	the 95-th percentile
quantile(h, probs = 0.1, 0.2, 0.99)	several quantiles at a time
summary(h)	5 percentiles and the mean
var(h)	the sample variance
sd(h)	the sample standard deviation

2.5.2 Coercion to numeric ectors

In 2.2.4, we have introduced how to combine values of different types into a vector by using the coercion rule in R, and the most important usage of coercion rule is to covert logical values into numbers. (The coercion rule will unify all values into the most complex one, and the order of complexity from simple to complex is logical < numeric < character.)

If you want to combine logical values and numbers into a vector, `TRUE` will be converted to 1 and `FALSE` will be converted to 0. Let's take a look at an example,

```
a <- c(TRUE, TRUE, 10, 07, FALSE)
a
```

```
#> [1]  1  1 10  7  0
```

Here, `a` is a numeric vector with five numbers. Then you can apply all the statistical functions introduced above on `a`.

2.5.3 Character vectors

Compared with numeric vectors, there are much less things you can do on character vectors. For character vectors, you can also apply `summary()`.

```
animals <- rep(c("sheep", "pig", "monkey"), 3:1)
summary(animals)
```

From the result, you can see that the `summary()` function will only tell you the vector length (6 elements) and vector type (character vector), much less useful than the case for numeric vectors.

2.5.4 Logical vectors

What if we apply `summary()` on logical vectors?

```
logic <- rep(c(T,F,T), 3:1)
summary(logic)
```

Similar to character vectors, you can get the vector type, which is `logical` here. You also get a frequency table for the times `FALSE` and `TRUE` appear in the vector.

2.5.5 Exercise

You can run the following code to do the exercise.

```
r02pro(2)
```

2.6 Comparisons, Vector Subsetting & Change values

By now, you have been familiar with the simplest R object—vector, and you have learnt how to create vectors and how to apply functions on vectors. In this section, we will introduce some new operations on vectors.

2.6.1 Comparisons on vectors of the same type

The first operation we want to introduce is making comparisons between two vectors of the **same type**. Similar to the arithmetic operations between two numeric vectors in Section 2.2, we normally want to compare two vectors of the same length, but we can also compare two vectors of different length according to the recycling rule in R.

a. Compare two vectors of the same length

If two vectors are of the same length, the comparison is done elementwisely, just like the arithmetic operations in Section 2.2.

Let's take numeric vectors for example, firstly you need to create a numeric vector `x` and make comparison to another numeric vector `2` to see whether the value of `x` is smaller than 2 or not.

```
x <- 3
x < 2
```

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

Since 3 is bigger than 2, you will get `FALSE` obviously!

In addition to the less than sigh `<`, you can try other symbols to do comparisons.

```
x < 2      #less
x <= 2     #less or equal
x > 1      #bigger
x >= 1     #bigger or equal
x == 3     #equal
x != 3     #not equal
```

Notice that if you want to see whether two vectors have the same value, you have to use two equal signs together, which is `==`, to do comparisons. And you can use an exclamation mark together with a equal sign, which is `!=`, to see whether two vectors have different values.

In 2.1.2, you have learned that R will get the result of an expression firstly if its values are unintuitive. That's because the result will show the **object type** and **values** of this expression. Here, you may notice that you will get `TRUE` or `FALSE` as the result from the codes above. Since `TRUE` and `FALSE` are logical values (there are no pairs of double quotes around `TRUE`s and `FALSE`s when you use them as logical values), you will know that all the objects above are logical vectors. Of course you can assign the value to a name for future use. Let's take `x > 1` for example

```
class(x > 1)
length(x > 1)
big1 <- x > 1
big1
class(big1)
```

So `x > 1` and `big1` are both logical vectors with one element.

Also, you can create two length>1 numeric vectors `y` and `z` with the same length, then you can do comparisons between them.

```
y <- c(3,5,7,5,3)
z <- c(2,6,7,6,3)
y > z
```

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

From the result, you can see that `y > z` is a length-5 logical vector. The values of `y > z` are obtained by making comparisons between the corresponding elements in these two vectors.

```
big2 <- y > z
big2
class(big2)
which(big2)
```

If you assign values of `y > z` to a name, you will create a logical vector `big2`. Here, the `which()` function can help you get the location of `TRUE` values, so you will get the result of `1` for `big2`.

You can also compare two character vectors. When making comparisons between character vectors, people usually use two equal signs `==` to see whether two corresponding elements have the same value. Let's first create two character vectors with the same length, then use `==` to compare them. Since the expression of comparison is a logical vector, you can do assignment operation to create a new logical vector `same1` and get the locations of `TRUE` values.

```
animals <- c("pig", "monkey", "pig")
zoo <- c("sheep", "monkey", "pig")
same1 <- animals == zoo
which(same1)
```

Comparisons between logical vectors work similarly as character vectors, people also usually use `==` to compare corresponding elements in logical vectors. Here are the examples.

```
logi1 <- c(TRUE, FALSE, FALSE)
logi2 <- c(TRUE, TRUE, TRUE)
same2 <- logi1 == logi2
which(same2)
```

b. Compare between one vector with length > 1 and another vector with length 1

The recycling rule also works for the comparison operation in R, similarly you need to use two vectors of the same type. If you have one vector with length 1, you can compare the value of it to the values of another vector with more than one element one by one, then you will get a logical vector as the result. The length of the logical vector will be the same as that of the longer vector. Here are some examples.

```
y != x
animals == "pig"
log1 == TRUE
```

```
#> Error in eval(expr, envir, enclos): object 'log1' not found
```

Of course you can assign the values of results to names and get locations of `TRUE` values by using the `which()` function. Try it by yourself!

2.6.2 Comparisons on vectors of different types

You can also make comparisons between two vectors of **different types**. Of course this operation also follow the rules that vectors are of the same length or comparing vectors of different length according to the recycling rule. But how can we compare vectors of different types?

Similar to the coercion rule in 2.2.4, values of corresponding elements will be unified into the more complex one when making comparisons between these two vectors. The order of complexity from simple to complex is still logical < numeric < character. Let's try to compare between a numeric vector and a logical vector,

```
a <- c(-1, 0, 1)
b <- c(TRUE, FALSE, TRUE)
a == b
```

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

Then you will get a length-3 logical vector from the result. In `a == b`, the first element is obtained by using `==` to compare `-1` and `TRUE`, then R will convert `TRUE` to `1` and make comparison between `-1` and `1` since numbers are more complex than logical values. The second and the third elements are also obtained in the same way. This is the most commonly use of comparisons between vectors of different types.

Of course you can make comparisons between vectors of other combinations, but we won't introduce in our book, you can try some by yourself!

2.6.3 Vector subsettings

Sometimes we may want to extract particular values from a vector, then the extracted values will constitute a new vector, which is the subset vector of the original vector. This process is called vector subsetting, and the subset vector will be the **same type** as the original one.

In this part, we will introduce two common ways to do vector subsettings in R, before we get started, let's create a vector which will be used throughout this part.

```
h <- c(3,1,4,2,90)
```

Then you can make a comparison between `h` and `2` and get a logical vector `big3`.

```
big3 <- h > 2
big3
```

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

a. Use logical vectors to do vector subsettings

Firstly, we will introduce the way of using logical vectors to do vector subsettings. You need to use a pair of square brackets `[ ]` after a vector, then put a logical vector with the **same length** as the previous one inside the square brackets. Here is an example,

```
h[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
```

```
#> [1]  3  4 90
```

From the result, you can see that you will get the values from `h` with the same positions of `TRUE`s. Since `3`, `4` and `90` are parts of the values of `h`, the vector composed of these three values is a subset vector of `h`.

Then you may notice that `big3` and `h > 2` have the same values as `c(TRUE, FALSE, TRUE, FALSE, TRUE)`, so you can also put `big3` or `h > 2` into `[ ]`, and you will get a numeric vector with `3`, `4` and `90` as values.

```
h[c(TRUE, FALSE, TRUE, FALSE, TRUE)]
h[big3]
h[h > 2]
```

If you create a character vector `home` and compare it to `"pig"`, you will get another logical vector `same3`. Let's try to use `same3` to do vector subsetting on `h`.

```
home <- c("pig", "monkey", "pig", "monkey", "pig")
same3 <- home == "pig"
h[same3]
```

```
#> [1]  3  4 90
```

Awesome! You still get the result of `3`, `4` and `90`! As a result, only if the logical vectors you used have the same values, you will get the same result after doing vector subsettings.

Of course you can do vector subsettings on character vectors or logical vectors, notice that the result will be the same type as the original one. Try the following code by yourself.

```
home[same3]
home[big3]
lg <- c(TRUE, FALSE, FALSE, FALSE, TRUE)
lg[same3]
lg[big3]
```

b. Use indices to do vector subsettings

Next, we will introduce how to use indices to do vector subsettings. To achieve this goal, you need to put a numeric vector into `[ ]`, for example,

```
h[c(2,4)]  #return values of the 2nd and 4th elements
```

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

Then you will get values of the 2nd and 4th elements in `h`. If you add a minus sign `-` before the numeric vector, you will get values except the 2nd and 4th elements in `h`.

```r
h[-c(2,4)]  #return values except the 2nd and 4th elements
```

```
#> [1]  3  4 90
```

Similar to use logical vectors to do vector subsettings, only if the numeric vectors have the same values, you will get the same result after doing vector subsettings.

```r
indices <- c(2,4)
h[indices]
```

Also, you can get subset vectors of character vectors or logical vectors according to indices.

```r
home[indices]
lg[indices]
```

2.6.4 Change values in named vectors

a. Change all values in subsets of vectors

Now you must be curious about why do we need to get subsets of vectors. Firstly let's review values of vector `h` and get a subset of it. (There are two ways to do vector subsettings, here we choose to use indices to get the subset)

```r
h <- c(3,1,4,2,90)
h[c(2,4)]
```

Obviously you will get a numeric vector with `1` and `2` as the values. Then we will introduce how to change values in subsets of `h` and put it back into `h` without changing other values, this is a very important usage of doing vector subsettings. You just need to assign new values to the subset, then you can verify the values of `h`, here is the example,

```r
h[c(2,4)] <- 10
h
```

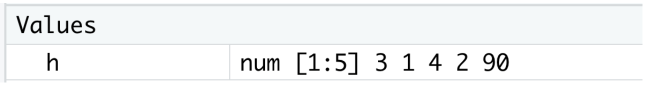
```
#> [1]  3 10  4 10 90
```

From the result, you can see that only `1` and `2` have been changed to `10`, which means you have successfully change the parts of `h`!

b. Define the vector again

Another way to change values in named vectors is to do object assignment again on this vector, then you can change all values of it without changing the name.

In Section 2.1, you have learned about checking all the named objects and their values in the environment. So let's review values of vector **h** from this panel together.



```
Values
  h          num [1:5] 3 1 4 2 90
```

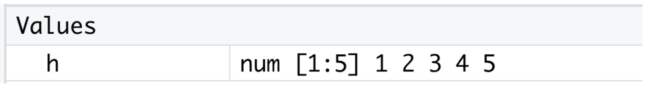
Figure 2.5: Values of h (1)

Now we all know that **h** is a numeric vector with 5 values. Then let's try to do an object assignment again, this time you can assign different values to **h** and see what will happen to **h**.

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

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

Then you can see that the values of **h** have been changed to the new ones! Another easier way to verify values of **h** is from the environment, so it is a good habit to monitor the environment from time to time to make sure everything look fine.

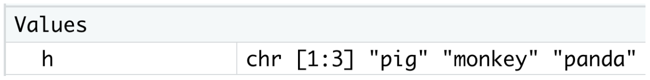


```
Values
  h          num [1:5] 1 2 3 4 5
```

Figure 2.6: Values of h (2)

You can assign any values to **h** as you want, then **h** may change the vector type or even the object type according to the values assigned. By running the following code, **h** will be a character vector with three strings.

```r
h <- c("pig", "monkey", "panda")
```



```
Values
  h          chr [1:3] "pig" "monkey" "panda"
```

Figure 2.7: Values of h (3)

If you assign values of the subset to a name, you will create a new named vector. Now **hs** is not the subset of **h**, it is a vector with the same value as the subset. If you assign diffrent value(s) to **hs**, there will be no change on **h**.

```
h <-  (3,1,4,2,90)
hs <- h[ (2,4)]
hs <- 10
h
```

2.6.5 Exercise

You can run the following code to do the exercise.

```
r02pro(2)
```

2.7 Comparisons, Vector Subsetting & Change values

By now, you have been familiar with the simplest R object—vector. You have learnt how to create vectors and how to apply functions on vectors. In this section, we will introduce some new operations on vectors.

2.7.1 Comparisons on vectors of the same type

The first operation we want to introduce is making comparisons between two vectors of the **same type**. Similar to the arithmetic operations between two numeric vectors in Section 2.2, we normally want to compare two vectors of the same length, but we can also compare two vectors of different length according to the recycling rule in R.

a. Compare two vectors of the same length

If two vectors are of the same length, the comparison is done elementwisely, just like the arithmetic operations in Section 2.2.

Let's take numeric vectors for example, firstly you need to create a numeric vector `x` and make comparison to another numeric vector 2 to see whether the value of `x` is smaller than 2 or not.

```
x <- 3
x < 2
```

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

2.7.2 Exercise

You can run the following code to do the exercise.

```r
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 2.5.

```
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 `NA`s 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 it 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.

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

```
#> 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.

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

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

```
str(kit_qual_ordered_factor)
```

```
#>  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.

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

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.

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

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)
```

`bookdown-demo_files/figure-latex/unnamed-chunk-181-1.pdf`

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 = "re
```

`bookdown-demo_files/figure-latex/unnamed-chunk-182-1.pdf`

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.

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

`bookdown-demo_files/figure-latex/unnamed-chunk-183-1.pdf`

Aside from the expected scatterplot, you can see a warning message "Removed 1 rows containing missing values (geom_point)." This indicate 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.

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

`bookdown-demo_files/figure-latex/unnamed-chunk-184-1.pdf`

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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-186-1.pdf

Now, let's see how to set **global aethetics** 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")
```

bookdown-demo_files/figure-latex/unnamed-chunk-187-1.pdf

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)
```

bookdown-demo_files/figure-latex/unnamed-chunk-188-1.pdf

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 =
```

bookdown-demo_files/figure-latex/unnamed-chunk-189-1.pdf

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 =
```

bookdown-demo_files/figure-latex/unnamed-chunk-190-1.pdf

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 =
```

bookdown-demo_files/figure-latex/unnamed-chunk-191-1.pdf

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 = h
```

bookdown-demo_files/figure-latex/scatterplot-color-1.pdf

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)
```

bookdown-demo_files/figure-latex/unnamed-chunk-192-1.pdf

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)
```

bookdown-demo_files/figure-latex/unnamed-chunk-193-1.pdf

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),
```

bookdown-demo_files/figure-latex/unnamed-chunk-194-1.pdf

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
```

bookdown-demo_files/figure-latex/unnamed-chunk-195-1.pdf

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 = h
```

bookdown-demo_files/figure-latex/unnamed-chunk-196-1.pdf

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
```

bookdown-demo_files/figure-latex/unnamed-chunk-197-1.pdf

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 = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price, color = heat_qual),
```

bookdown-demo_files/figure-latex/unnamed-chunk-198-1.pdf

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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-199-1.pdf

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 lay 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
```

bookdown-demo_files/figure-latex/unnamed-chunk-200-1.pdf 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
```

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

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 = lo
```

bookdown-demo_files/figure-latex/unnamed-chunk-203-1.pdf 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 = l
```

You will 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 > 1
```

bookdown-demo_files/figure-latex/unnamed-chunk-205-1.pdf

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
```

bookdown-demo_files/figure-latex/unnamed-chunk-206-1.pdf

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.í

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'
```

bookdown-demo_files/figure-latex/unnamed-chunk-207-1.pdf

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 = FA
```

In addition to the default loess method for smoothline fit, `geom_smooth()` also provides other smoothing moethods. 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
```

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

bookdown-demo_files/figure-latex/unnamed-chunk-210-1.pdf

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 = sahp) + geom_smooth(mapping = aes(x = liv_area, y = sale_price, group = good_qual))
```

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

bookdown-demo_files/figure-latex/unnamed-chunk-212-1.pdf

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'
```

bookdown-demo_files/figure-latex/unnamed-chunk-213-1.pdf

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.ß

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
```

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

bookdown-demo_files/figure-latex/unnamed-chunk-214-1.pdf

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 = go
```

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

bookdown-demo_files/figure-latex/unnamed-chunk-215-1.pdf

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

bookdown-demo_files/figure-latex/unnamed-chunk-216-1.pdf

When you try 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, you 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'
```

bookdown-demo_files/figure-latex/unnamed-chunk-217-1.pdf

3.5 Multiple geoms and Aesthetics

So far, you have learned to create scatterplots using `geom_point()` and smoothline 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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-218-1.pdf

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

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

bookdown-demo_files/figure-latex/unnamed-chunk-218-2.pdf

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_s
```

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

bookdown-demo_files/figure-latex/unnamed-chunk-219-1.pdf

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_s
```

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

bookdown-demo_files/figure-latex/unnamed-chunk-220-1.pdf

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'
```

bookdown-demo_files/figure-latex/unnamed-chunk-221-1.pdf

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'
```

bookdown-demo_files/figure-latex/unnamed-chunk-222-1.pdf

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(mappi
```

bookdown-demo_files/figure-latex/unnamed-chunk-223-1.pdf

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_
```

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)) + g
```

bookdown-demo_files/figure-latex/unnamed-chunk-225-1.pdf

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 using the same mapping in each geom as the global mapping, we can also extend or overwrite the global mapping in each geom.

3.6.1 Extend Global Aesthetic Mappings

In addition to the global aesthetic mappings, you can add additional mappings in each geom.

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

bookdown-demo_files/figure-latex/unnamed-chunk-226-1.pdf

Here, both `geom_point()` and `geom_smooth()` will first inherit the global mappings, then add the `shape` and `linetype` aesthetics, respectively. As you can see from the legend, there are some missing values for the variable `gar_car`. If you want to remove the observations that has a missing `gar_car`, you can use the function `remove_missing()`.

```
ggplot(data = remove_missing(sahp, vars = "gar_car"), mapping = aes(x = lot_area, y = sale_price,
```

bookdown-demo_files/figure-latex/unnamed-chunk-227-1.pdf

In the `remove_missing()` function, we use the argument `vars = "gar_car"` to indicate that only the observaitons with a missing `gar_car` are removed. Note that you may be tempting to use the function `na.omit()` on the data `sahp`. However, it is not recommended as the function will remove all observations whenever there is missing value for one variable, even if the variable is not used in the visualization process.

3.6.2 Overwrite Global Aesthetic Mappings

When you add the same aesthetic mapping in the local geom as one of the global mappings, the corresponding local aesthetic mapping will be overwritten.

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

bookdown-demo_files/figure-latex/unnamed-chunk-228-1.pdf

Here, you can see that the `color` aesthetic mapping corresponding to the logical variable `gar_car > 1`.

It is worth to note the difference between *global aesthetics* (introduced in Section 3.3) and the *global aesthetic mappings* introduced here. The *global aesthetics* set the aesthetic for all points/lines on the graph while the *global aesthetic mapping* represents the mapping which will be passed to all geoms. Although we can set *global aesthetic mappings*, it is impossible to set *global global aesthetics*. Let's try to set all points and lines to be red.

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

bookdown-demo_files/figure-latex/unnamed-chunk-229-1.pdf

As you can see the `color = "red"` argument was ignored during the plotting process. To achieve this, you need to use `color = "red"` as global aesthetics for both geoms as below.

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

bookdown-demo_files/figure-latex/unnamed-chunk-230-1.pdf

3.6.3 Mix

As you may have expected, it is straightforward to mix global aesthetic mappings, local aesthetic mappings, and global aesthetics for each geom.

```
ggplot(data = remove_missing(sahp, vars = "kit_qual"), mapping = aes(x = liv_area, y =
```

bookdown-demo_files/figure-latex/unnamed-chunk-231-1.pdf

Note that we again use the `remove_missing()` function to remove observations with missing `kit_qual`.

3.7 Line Plots via geom_line()

In this section, we will introduce the **line plot**, which is very useful for visualizing trends and often used in time series.

3.7.1 An Introduction to Line Plots

In the `sahp` dataset, it would be interesting to generate a plot showing the trend of the sale price as a function of the sold date of the house. To do this, we can use the `geom_line()` function.

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

bookdown-demo_files/figure-latex/unnamed-chunk-232-1.pdf

The `geom_line()` works by first getting the location of the value pairs of `dt_sold` and `sale_price`. Then, connect the points in the order of the variable on the x axis, i.e. `dt_sold` in this example.

To get a better view on how `geom_line()` works, let's focus on the houses that were sold before 2007.

```
sahp_2006 <- sahp[format(sahp$dt_sold, "%Y") < 2007, ] #all houses sold before 2007
ggplot(data = sahp_2006) + geom_line(mapping = aes(x = dt_sold, y = sale_price))
```

bookdown-demo_files/figure-latex/unnamed-chunk-233-1.pdf

Next, we add the scatterplot onto the plot using the global mappings.

```
ggplot(data = sahp_2006, mapping = aes(x = dt_sold, y = sale_price)) + geom_line() + geom_point()
```

bookdown-demo_files/figure-latex/unnamed-chunk-234-1.pdf

3.7.2 Aesthetics in Line Plots

As expected, we can also map variables to aesthetics in line plots.

```
ggplot(data = sahp) + geom_line(mapping = aes(x = dt_sold, y = sale_price, color = kit_
```

bookdown-demo_files/figure-latex/unnamed-chunk-235-1.pdf

Here, the observations are divided into groups according to the value of `kit_qual` and separate line plots are generated for each group, representing as different colors. In addition, we can also map variables to the `linetype` aesthetic as in the `geom_smooth()` function.

```
ggplot(data = sahp) + geom_line(mapping = aes(x = dt_sold, y = sale_price, linetype = k
```

In addition to mapping aesthetics, you can also set global aesthetics as before.

```
ggplot(data = sahp) + geom_line(mapping = aes(x = dt_sold, y = sale_price ), color = "b
```

bookdown-demo_files/figure-latex/unnamed-chunk-237-1.pdf

3.8 Add Auxiliary Lines

Having learned how to generate scatterplots, smoothline fits, and line plot, it is sometimes helpful to add auxiliary lines to existing plots to provide additional information.

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

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

bookdown-demo_files/figure-latex/unnamed-chunk-238-1.pdf

Looking at the scatterplot, it maybe helpful to add a horizontal line. To do this, you can use the `geom_hline()` function with argument `yintercept` specifying the value on the y-axis.

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

bookdown-demo_files/figure-latex/unnamed-chunk-239-1.pdf

Here, a horizontal line at 300 is added to the scatterplot.

You can also add both vertical lines and horizontal lines to the same plot.

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

bookdown-demo_files/figure-latex/unnamed-chunk-240-1.pdf

In addition to adding vertical lines and horizontal lines, you can also add any line with the `geom_abline()` function. We know a line can be represented as a function $y = a + b \times x$, where a is the intercept and b is the slope. In the `geom_abline()`, you can generate such a line by specifying the `slope` and `intercept` arguments.

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

bookdown-demo_files/figure-latex/unnamed-chunk-241-1.pdf

3.9 Jitter and Count Plots

We have seen that scatterplot is a useful tool to visualization the relationship between two continuous variables. You may be wondering what will happen if we use it on two discrete variables.

3.9.1 Overplotting

As an example, let's generate a scatterplot between `kit_qual` and `heat_qual`.

```
library(r02pro)
library(tidyverse)
ggplot(data = sahp) + geom_point(mapping = aes(x = kit_qual, y = heat_qual)) #overplot
```

bookdown-demo_files/figure-latex/unnamed-chunk-242-1.pdf

From the plot, you may immediately realize that there are many overlapping data points. Actually, there will be at most 16 possible distinct points on the plot since both variables have 4 categories. This phenomenon is called **overplotting**. Overplotting is not desirable since it hides useful information about the joint distribution. For example, we don't know which value pairs out of the 16 possibilities appear more frequently in the data.

To solve the overplotting issues, we introduce two solutions, namely jittering and count plots.

3.9.2 Jittering

The first method for solving the overplotting issue is to add a small random perturbation to all datapoints, i.e., **jittering**. You can use the `geom_jitter()` function which works by first perturb the data points and then generate a scatterplot.

```
ggplot(data = sahp) + geom_jitter(mapping = aes(x = kit_qual, y = heat_qual))
```

bookdown-demo_files/figure-latex/unnamed-chunk-243-1.pdf

For the jittered plot, we can clearly see which pair of `kit_qual` and `heat_qual` have more observations. By default, the perturbation will be performed both vertically and horizontally with the same amount of 40% of the resolution of the data. To customize the amount of jittering, you can specify the arguments `width` as the amount of horizontal jittering and `height` as the amount of vertical jittering in the unit of the resolution of the data. To turn off the horizontal jittering, you can specify `width = 0`.

```
ggplot(data = sahp) + geom_jitter(mapping = aes(x = kit_qual, y = heat_qual),  width = 0.1, heigh
```

bookdown-demo_files/figure-latex/unnamed-chunk-244-1.pdf

3.9.3 Counts Plots

When we want to visualize the distribution of a pair of discrete variables, another method to solve the overplotting issue is the **counts plot**, which uses circles of different sizes to represent the frequency of each value pair. You can use the function `geom_count()` to generate a counts plot.

```
ggplot(data = sahp) + geom_count(mapping = aes(x = kit_qual, y = heat_qual))
```

bookdown-demo_files/figure-latex/unnamed-chunk-245-1.pdf

From this plot, you can clearly tell the frequency of each value pair by the legend showing the relationship between the size of the circle and the count.

3.10 Bar Charts via geom_bar()

For visualizing the relationship between two continuous variables, we have been learned various kinds of plots, including scatterplot (Sections 3.2 and 3.3), smoothline fit (Section 3.4), and line plot (Section 3.7).

In this section, we will introduce a new kind of plot called **bar chart**. A bar chart uses rectangular bars with heights or lengths proportional to the values they represent, in order to visualize a discrete variable.

3.10.1 An Introduction to Bar Chart

In the `sahp` dataset, you may be interested in the distribution of kitchen quality of the house (denoted in the `kit_qual` variable). To generate a bar chart, you can use the function `geom_bar()`.

```
library(ggplot2)
library(r02pro)
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual))
```

bookdown-demo_files/figure-latex/unnamed-chunk-246-1.pdf

In the bar chart, the x-axis displays different values of `kit_qual`, and the y-axis displays the number of observations with each `kit_qual` value. To verify this, let's check how many house have `kit_qual` equals Excellent.

```
sum(sahp$kit_qual == "Excellent")
```

```
#> [1] 14
```

We can see the answer 14 matches the value on the bar chart.

You may have noticed that the y-axis count is not a variable in `sahp`! This is also the reason that we don't need to specify the `y` argument in the `aes()` function. In this sense, bar charts are very different from many other graphs like scatterplots, which plot the raw values of datasets.

Sometimes, we would like to find out the proportion for each value of `x`, then we can display a bar chart of proportion, rather than count. To do this, we need to add `y = stat(prop)` and `group = 1` ad additional arguments in the `aes()` function. The `stat(prop)` is a statistical function used to calculate proportions. The `group = 1` implies that all the observations belong to one single group when calculating the proportions. We will try to set `group` to another variable in the next part.

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual, y = stat(prop), group = 1))
```

bookdown-demo_files/figure-latex/unnamed-chunk-248-1.pdf

3.10.2 Reordering Bars in Bar Charts

In our bar chart example, the bars are ordered alphabetically. Sometimes, we may want to reorder the bars according to certain criterion. ***a. Reorder in ascending/descending order of heights*** To order the bars in ascending/descending order of their heights, you can use the `fct_reorder()` function to reorder the factor.

```
library(forcats)
ggplot(data = sahp) + geom_bar(mapping = aes(x = fct_reorder(kit_qual, kit_qual, length))) #incre
```

bookdown-demo_files/figure-latex/unnamed-chunk-249-1.pdf

The above code reorders the bar chart to the increasing order of their heights. The `fct_reorder()` is a very powerful function used to record the levels of a factor. The function `fct_reorder(.f, .x, .fun = median)` has three arguments:

- `.f`: the discrete variable/factor to reorder
- `.x`: one variable
- `.fun`: the function to be applied to `.x`

The levels of `f` are reordered so that the values of `.fun(.x)` are in ascending order. In our example, we reorder the levels of `kit_qual` such that the length of `kit_qual` is in ascending order, i.e. the bars are in ascending order of their heights.

To reorder in descending order of heights, you can add and additional argument `.desc = TRUE` in the `fct_reorder()` function.

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = fct_reorder(kit_qual, kit_qual, length, .desc =
```

bookdown-demo_files/figure-latex/unnamed-chunk-250-1.pdf

b. Manual Reorder

In addition to reordering according to certain function values, you can use the function `fct_relevel()` to **manually reorder** the bars in a bar chart. In our example of `kit_qual`, perhaps a common thought is to order the levels from the worst quality to the best quality.

```
ggplot(data = sahp) +
  geom_bar(mapping = aes(x = fct_relevel(kit_qual, c("Average", "Fair", "Good", "Excell
```

bookdown-demo_files/figure-latex/unnamed-chunk-251-1.pdf

As you can imagine, the second argument of `fct_relevel()` contains the desired order of the factors, which will be reflected in the order of the bars.

3.10.3 Aesthetics in Bar Charts

As before, we can use aesthetics to control the appearance of bar charts.

First, let's look at a new aesthetic called **fill**, which fills the bar with different colors according to the value of the mapped variable (usually another discrete variable). Here, we want to look at the distribution of `kit_qual` for different values of `central_air`.

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual, fill = central_air))
```

bookdown-demo_files/figure-latex/unnamed-chunk-252-1.pdf

We can see that each bar is divided into stacked sub-bars with different colors. The different colors in each sub-bar correspond to the value of `central_air`.

And the height of each sub-bar represents the count for the cases with a particular value of `kit_qual` and another value of `central_air`.

Let's verify the first bar.

```
sum(sahp$kit_qual == "Average" & sahp$central_air)
```

```
#> Error in sahp$kit_qual == "Average" & sahp$central_air: operations are possible only for numer
```

Clear, the result matches the blue portion of the first bar.

When we map a variable to the `fill` aesthetic, the appearance of different subbars can be customized using the `position` argument as a global aesthetic in `geom_bar()`.

a. stacked bars

The default value of the `position` argument is `"stack"`, which generated a collection of stacked bars with different colors.

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual, fill = central_air), position = "stack
```

b. dodged bars

Using the stacked bars, it is sometimes difficult to compare the counts for different subbars. To make the comparison easier, you can use `position = "dodge"`.

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual, fill = central_air), position = "dodge
```

bookdown-demo_files/figure-latex/unnamed-chunk-255-1.pdf

Now, the plot places the sub-bars beside one another, which makes it easier to compare individual counts for each combination of `kit_qual` and `central_air`.

c. filled bars

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual, fill = central_air), position = "fill'
```

bookdown-demo_files/figure-latex/unnamed-chunk-256-1.pdf

`position = "fill"` works like stacking, but makes each set of stacked bars the same height. The `y` axis should be labeled as "proportion" rather than "count". This makes it easier to compare proportions of different values of `central_air` for different values of `kit_qual`. For example, we can see that the proportion of `central_air = TRUE` is much higher for `kit_qual = "Good"` than that for `kit_qual = "Fair"`.

Lastly, you can also use the polar coordinates.

```
ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual)) + coord_polar()
```

bookdown-demo_files/figure-latex/unnamed-chunk-257-1.pdf

3.11 Customization of x and y Axes

In this section, we would like to digress a little bit to learn some possible customizations of the x and y axes.

First, let's review 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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-258-1.pdf

3.11.1 Customizing the Breaks on the x and y Axes

In the above plot, the breaks on the x axis are 1000, 2000, and 3000. The breaks on the y axis are 100, 200, 300, 400, and 500. Sometimes, we may want to customize the breaks, e.g. to show a finer scale. To do this, we can use the `scale_x_continous()` and `scale_y_continuous()` functions. Both functions take an argument called `breaks` which is a numeric vector specifying the desired breaks on the x and y axes.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) +
  scale_x_continuous(breaks = seq(from = 500, to = 3500, by = 500)) +
  scale_y_continuous(breaks = seq(from = 50, to = 600, by = 50))
```

bookdown-demo_files/figure-latex/unnamed-chunk-259-1.pdf

In this example, we have customized the breaks for `liv_area` to be a equally-spaced sequence from 500 to 3500 with increment 500, and the breaks for `sale_price` to be another equally-spaced sequence from 50 to 600 with increment 50.

When the specified breaks do not cover the full range of the data, you will see the breaks changed but all the data points are still visible.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) +
  scale_y_continuous(breaks = seq(from = 300, to = 600, by = 50))
```

bookdown-demo_files/figure-latex/unnamed-chunk-260-1.pdf

On the other hand, if the specified breaks go beyond the value of the data, `ggplot()` will only show the breaks values within the data range.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) +
  scale_x_continuous(breaks = seq(from = 500, to = 5000, by = 500)) +
  scale_y_continuous(breaks = seq(from = 200, to = 800, by = 50))
```

bookdown-demo_files/figure-latex/unnamed-chunk-261-1.pdf

3.11.2 Zoom In to a Specific Region of the Data

Sometimes, you want to zoom in to the specific region of the data to see a finer detail. To do this, you can use the `coord_cartesian()` function with arguments `xlim` and `ylim` for specifying the desired region.

Let's say we want to focus on the houses with `liv_area` between 1000 and 2000.

```r
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))  +
  coord_cartesian(xlim = c(1000, 2000))
```

bookdown-demo_files/figure-latex/unnamed-chunk-262-1.pdf

Let's narrow down further to only the houses with `liv_area` between 1000 and 2000, and `sale_price` between 200 and 300.

```r
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price))  +
  coord_cartesian(xlim = c(1000, 2000), ylim = c(200, 300))
```

bookdown-demo_files/figure-latex/unnamed-chunk-263-1.pdf

3.11.3 Plot with Transformed Variables and Log-scale Plot

In some applications, instead of using the original variables, you may want to generate a plot with certain transformations of them.

In our scatterplot, maybe we want to change the y-axis to the logarithm of the sale price.

```r
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = log10(sale_price)))
```

bookdown-demo_files/figure-latex/unnamed-chunk-264-1.pdf

The working mechanism is to first generate a temporary variable `log10(sale_price)` on the fly, and then generate a scatterplot between `liv_area` and the transformed variable.

For this particular `log10()` transformation, an alternative is to generate a **Log-scale Plot** by setting the `trans` argument in the `scale_y_continous()` function. The `log-scale plot` is a popular way for displaying numerical data over a very wide range of values in a compact fashion.

The default value of `trans` is `"identity"` meaning no transformation. There are many different `trans` choices including `log`, `exp`, `log10`, `sqrt`, and others.

```
ggplot(data = sahp) + geom_point(mapping = aes(x = liv_area, y = sale_price)) +
  scale_y_continuous(trans = "log10")
```

bookdown-demo_files/figure-latex/unnamed-chunk-265-1.pdf

This works in a very different way by change the scale of the y-axis.

3.12 Histograms

In Section 3.10, we learned how to use `geom_bar()` to generate bar charts for visualizing the distributions of discrete variables. You may be wondering, how about visualizing continuous variables? One popular plot is called **histograms**.

Let's again use the `sahp` housing price dataset.

3.12.1 Using the `hist()` function

To generate a histogram, you can simply use `hist()` with the variable as the argument.

```
library(r02pro)
hist(sahp$sale_price)
```

bookdown-demo_files/figure-latex/unnamed-chunk-266-1.pdf

On the x-axis, the histogram displays the range of values for the sale price. Then, the histogram divides the x-axis into bins with equal width, and a bar is erected over the bin with the y-axis showing the corresponding number of observations (called Frequency on the y-axis label).

3.12.2 Using the `geom_histogram()` function

In addition to using the `hist()` function in base R. The `geom_histogram()` function in the **ggplot2** package provides richer functionality. Let's take a quick look.

```
library(ggplot2)
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price))
```

```
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

bookdown-demo_files/figure-latex/unnamed-chunk-267-1.pdf

We could see a message, saying "`stat_bin()` using `bins = 30`" which implies the histogram has 30 bins by default. Next, we introduce three different ways to customize the bins.

a. Use aesthetic `bins`.

We can change the number of bins with the global aesthetic `bins`.

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price), bins = 5)
```

bookdown-demo_files/figure-latex/unnamed-chunk-268-1.pdf

We can see that the histogram now has 5 bins and each bin has the same width.

b. Use aesthetic `binwidth`.

And from the message, another way to change the number of bins is to specify the `binwidth`, which is another global aesthetic.

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price), binwidth = 1e2)
```

bookdown-demo_files/figure-latex/unnamed-chunk-269-1.pdf

c. Manually set the bins.

If desired, you can manually set the bins via the `breaks` argument in the `geom_histogram()` function.

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price), breaks = seq(from = 0, to = 6
```

bookdown-demo_files/figure-latex/unnamed-chunk-270-1.pdf

The `breaks` argument is a numeric vector specifying how the bins are constructed. Let's verify the height of the first bin.

```
#verify the first bin count
sum(sahp$sale_price < 100)
```

3.12.3 Aesthetics in `geom_histogram()`

Next, we introduce the aesthetics in histograms, which are very similar to those in bar charts. For example, we can map a variable to the fill aesthetic.

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price, fill = house_style), bins = 5)
```

bookdown-demo_files/figure-latex/unnamed-chunk-272-1.pdf

We can see that like the bar chart, the bar for each bin is now divided into sub-bars with different colors. The different colors in each sub-bar correspond to the values of `house_style`. And the height of each sub-bar represents the count for the cases with the `sale_price` in this particular bin and the specific value of `house_style`. Just like `geom_bar()`, we have a global aesthetic called `position`, which does position adjustment for different sub-bars. The default position value is again `"stack"` if you don't specify it.

a. Stacked bars

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price, fill = house_style)
```

As expected, we get the same histogram as before.

b. Dodged bars

The second option for position is `"dodge"`, which places the sub-bars beside one another, making it easier to compare individual counts for the combination of a bin of `sale_price` and `house_style`.

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price, fill = house_style)
```

`bookdown-demo_files/figure-latex/unnamed-chunk-274-1.pdf`

c. Filled bars

Another option for optional is `position = "fill"`. It works like stacking, but makes each set of stacked bars the same height.

```
ggplot(data = sahp) + geom_histogram(mapping = aes(x = sale_price, fill = house_style)
```

`bookdown-demo_files/figure-latex/unnamed-chunk-275-1.pdf`

Just like `geom_bar()`, the y axis may be labeled as "proportion" rather than "count", to be more precise. This makes it easier to compare proportions of different values of cut for different bins of price. For example, we can see that the proportion of `house_style = "1.5Fin"` is highest for the houses with sale price less than 100.

3.12.4 Density Estimate Using Histograms

In addition to using histograms to visualize the distribution of a discrete variable, you can also construct a density density of variable using a proper normalization. To generate such a density estimate, you can add `y = ..density..` as a mapping in the `aes()` function in `geom_historgram()`. Let's see an example as below.

```
ggplot(data = sahp) +  geom_histogram(aes(x = sale_price, y = ..density..), breaks = seq(from = 
```

bookdown-demo_files/figure-latex/unnamed-chunk-276-1.pdf

Here, we added the `breaks` arguments on the bins as well as on the y-axis.

Let's try to calculate the height of the first bar together. We know that the total area of the bars is 1, agreeing with the definition of density. First, we get the area of the first bar. Then we divide it by the width to get the height.

```
sale_price_no_na <- na.omit(sahp$sale_price)
sum(sale_price_no_na < 1e2)/length(sale_price_no_na) #area of the first bar
```

```
#> [1] 0.1036585
```

```
sum(sale_price_no_na < 1e2)/length(sale_price_no_na) /1e2 #height of the first bar
```

```
#> [1] 0.001036585
```

You can see that the height matches the `y` axis for the first bar.

3.13 Density Plots

In Section 3.12, we have learned to use `geom_histogram()` as a way to visualize the distribution of a continuous variable. In addition, we can also use it to generate a piece-wise constant estimate of the probability density function. Today, we will introduce another visualization method for continuous data, namely the **density plots**. First, let's review the `geom_histogram()` for estimating the density function.

```
library(ggplot2)
library(r02pro)
ggplot(data = sahp) +  geom_histogram(aes(x = sale_price, y = ..density..))
```

```
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

bookdown-demo_files/figure-latex/unnamed-chunk-278-1.pdf

There are 30 bins by default. You may notice that this density estimate is not smooth, sometimes we may prefer a smoothed estimate. Then, we can use the `geom_density()` function to achieve this.

```
ggplot(data = sahp) + geom_density(aes(x = sale_price))
```

bookdown-demo_files/figure-latex/unnamed-chunk-279-1.pdf

This plot shows the so-called "kernel density estimate", a popular way to estimate the probability density function from sample. The density estimate can be viewed as a smoothed version of the histogram. We can combine the two plots together using global mapping.

```
ggplot(data = sahp, aes(x = sale_price)) +
  geom_histogram(aes(y = ..density..)) +
  geom_density(color = "red", size = 2)
```

```
#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

bookdown-demo_files/figure-latex/unnamed-chunk-280-1.pdf

Here, we added some global aesthetics in `geom_density()` to make the density plot red and the line width by setting `size = 2`. It is clear that the density plot is a useful alternative to the histogram for visualizing continuous data.

3.13.1 Aesthetics in Density Plots

Now, let's introduce some commonly used aesthetics for density plots.

a. Color

```
ggplot(data = remove_missing(sahp, vars = "oa_qual")) +  geom_density(aes(x = sale_price, color =
```

bookdown-demo_files/figure-latex/unnamed-chunk-281-1.pdf

Here, we divide the data into two groups according to the value of `oa_qual`, then generate separate density estimates with different colors for `oa_qual > 5` and `oa_qual <= 5`. The blue curve represents the density estimates for larger values of `oa_qual` while the red curve corresponds to that of the houses with smaller values.

b. Fill

Another way to generate different density estimates is to use the `fill` aesthetic. Let's see the following example.

```
ggplot(data = remove_missing(sahp, vars = "oa_qual")) +  geom_density(aes(x = sale_price, fill =
```

bookdown-demo_files/figure-latex/unnamed-chunk-282-1.pdf

The `fill` aesthetic also divides the data into groups according to `oa_qual`, then generate separate density estimates. The difference between `fill` and `color` aesthetics is that `fill` generates shaded areas below each density curve with different colors while `color` generates density curves with different colors. As we can see from the plot, there is a substantial overlap of the shaded areas. To fix this issue, we can change the transparency of the shades by adjusting the value of the `alpha` aesthetic.

```
ggplot(data = remove_missing(sahp, vars = "oa_qual")) +  geom_density(aes(x = sale_price, fill =
```

bookdown-demo_files/figure-latex/unnamed-chunk-283-1.pdf

We can now see both shaded areas in a clearly way.

c. Linetype

We can also use different linetypes for different curves.

```
ggplot(data = remove_missing(sahp, vars = "oa_qual")) +  geom_density(aes(x = sale_pri
```

bookdown-demo_files/figure-latex/unnamed-chunk-284-1.pdf

d. Global aesthetics

As usual, we can also set global aesthetics for `geom_density()` and combine it with the mapped aesthetics.

```
ggplot(data = remove_missing(sahp, vars = "oa_qual")) +  geom_density(aes(x = sale_pri
```

bookdown-demo_files/figure-latex/unnamed-chunk-285-1.pdf

Here, the `size` controls the width of the density curve.

3.14 Boxplots

So far, we have learned two ways to visualize a continuous variable, namely the histograms (Section 3.12) and density plots (Section 3.13). Now, we introduce another popular plot for visualizing the distribution of a continuous variable, namely the **boxplot**. Let's say we want to generate a boxplot for the variable `sale_price` in the `sahp` dataset.

3.14.1 Using the `boxplot()` function

To generate a boxplot, you can just use `boxplot()` with the variable as the argument.

```
library(r02pro)
sale_price <- na.omit(sahp$sale_price)
boxplot(sale_price)
```

```
bookdown-demo_files/figure-latex/unnamed-chunk-286-1.pdf
```

The boxplot compactly summarize the distribution of a continuous variable by visualizing five summary statistics (the median, two hinges, and two whiskers), and show all "outlying" points individually. All five summary statistics on the boxplot are related to the summary statistics we learned in Section 2.5. Let's first review the summary function and the inter quartile range (IQR).

```
summary(sale_price)
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    44.0   130.0   157.9   179.9   201.6   545.2
```

```
IQR(sale_price)
```

```
#> [1] 71.6125
```

Let's discuss the five lines on the boxplot.

- The **solid line** in the middle represents the median value, which is 157.95.
- The **lower solid line**, also known as the lower hinge, is the first quartile Q1 = 129.9625.
- The **upper solid line**, also known as the upper hinge, is the third quartile Q3 = 201.575.
- The **lower whisker** is the smallest observation value that is greater than or equal to Q1 - 1.5 * IQR. To find this value, we first calculate Q1 - 1.5 * IQR = 22.54375. Then, the smallest observation larger than 22.54375 is

```
lower_whisker_loc <- which.min(sale_price >= quantile(sale_price, 0.25) - 1.5 * IQR(sale_price))
sale_price[lower_whisker_loc]
```

```
#> [1] 130.5
```

- The **upper whisker** is the largest observation value that is smaller than or equal to Q3 + 1.5 * IQR. Similarly, the value is

```
upper_whisker_loc <- which.max(sale_price >= quantile(sale_price, 0.75) + 1.5 * IQR(sale_price))
sale_price[upper_whisker_loc]
```

```
#> [1] 344.133
```

To summarize, the five lines on the boxplot, from the top to bottom, are

- upper whisker (<= Q3 + 1.5*IQR)
- upper hinge (Q3)
- median (50-th percentile)
- lower hinge (Q1)
- lower whisker (>= Q1 - 1.5*IQR)

For the observations that are larger than the upper whisker or smaller than the lower whisker, the points are shown individually as **outliers**.

3.14.2 Using the `geom_boxplot()` function

As before, we will spend more time to discuss `geom_boxplot()` as it provides more functionality. Let's first create the boxplot for `sale_price`.

```
library(tidyverse)
ggplot(data = sahp) + geom_boxplot(aes(x = "", y = sale_price))
```

bookdown-demo_files/figure-latex/unnamed-chunk-290-1.pdf

Note that here we set `x = ""` since no information is needed on the x-axis.

In addition to the default summary statistics, we can add other values to the boxplot, for example, we can add the mean value to the plot.

```
ggplot(data = sahp, aes(x = "", y = sale_price)) + geom_boxplot() +
  geom_point(stat = "summary", fun = "mean", shape = 20, size = 4, color = "red")
```

bookdown-demo_files/figure-latex/unnamed-chunk-291-1.pdf

The `geom_point()` function will first calculate the mean hwy, and add it to the boxplot. Note that we used some global aesthetics for `geom_point()`.

3.14.3 Compare distributions in different groups

One common use of boxplot is to compare the distribution of a continuous variable in different groups. To do this, you just need to set the x-axis to be the discrete variable that encodes the different groups.

Let's say we want to compare the `sale_price` for houses with different `kit_qual`.

```
ggplot(data = sahp) + geom_boxplot(aes(x = kit_qual, y = sale_price))
```

bookdown-demo_files/figure-latex/unnamed-chunk-292-1.pdf

This plot shows the boxplots of `sale_price` for different values of `kit_qual` side-by-side, which makes the comparison of distributions straightforward.

Just like in bar charts, you may want to arrange the boxplots in a particular order. For example, to order the boxplots in ascending order of the `sale_price`, you can use

```
ggplot(data = remove_missing(sahp, vars = "sale_price")) + geom_boxplot(aes(x = fct_reorder(kit_q
```

bookdown-demo_files/figure-latex/unnamed-chunk-293-1.pdf

To order it by the mean `sale_price` in descending order, you can use

```
ggplot(data = remove_missing(sahp, vars = "sale_price")) + geom_boxplot(aes(x = fct_reorder(kit_q
```

bookdown-demo_files/figure-latex/unnamed-chunk-294-1.pdf

If you want to generate a flipped version of the boxplot, you can add `coord_flip()` to the `ggplot()` function. Actually, this works with any ggplot.

```
ggplot(data = sahp) + geom_boxplot(aes(x = kit_qual, y = sale_price)) +
  coord_flip()
```

bookdown-demo_files/figure-latex/unnamed-chunk-295-1.pdf

As an alternative, you can also switch the `x` and `y` arguments.

```
ggplot(data = sahp) + geom_boxplot(aes(x = sale_price, y = kit_qual))
```

bookdown-demo_files/figure-latex/unnamed-chunk-296-1.pdf

Now, you have learned how to compare the distributions of a continuous variable for different groups implied by a discrete variable. How about groups implied by a continuous variable? To do this, you can use the function `cut_width()` to convert a continuous variable to a discrete one by dividing the observations into different groups, just like in histograms. Let's try to convert the continuous variable `oa_qual` into a discrete one.

```
cut_width(sahp$oa_qual, width = 2)
```

```
#>   [1] (5,7]  (5,7]  (3,5]  (3,5]  (5,7]  (5,7]  (5,7]  (3,5]  (3,5]  (3,5]
#>  [11] (5,7]  (5,7]  (3,5]  (7,9]  (5,7]  (3,5]  (3,5]  (3,5]  (5,7]  (5,7]
#>  [21] (3,5]  (7,9]  (7,9]  <NA>   (5,7]  (3,5]  (3,5]  (3,5]  (3,5]  (7,9]
#>  [31] (7,9]  (7,9]  (5,7]  (7,9]  (5,7]  (3,5]  (5,7]  (5,7]  (3,5]  (3,5]
#>  [41] (9,11] (3,5]  (3,5]  (3,5]  (7,9]  (3,5]  (5,7]  (3,5]  (5,7]  (5,7]
#>  [51] (3,5]  (5,7]  (5,7]  (3,5]  (5,7]  (5,7]  (7,9]  (7,9]  (3,5]  (5,7]
#>  [61] (5,7]  (5,7]  (5,7]  (5,7]  (3,5]  (3,5]  (5,7]  (5,7]  (3,5]  (5,7]
#>  [71] (5,7]  (3,5]  (7,9]  (5,7]  (3,5]  (7,9]  (7,9]  (3,5]  [1,3]  (5,7]
#>  [81] (5,7]  (5,7]  (3,5]  (5,7]  (5,7]  (3,5]  (3,5]  (5,7]  (3,5]  (3,5]
#>  [91] (7,9]  (5,7]  (5,7]  (5,7]  (5,7]  (3,5]  (5,7]  (3,5]  (9,11] (5,7]
#> [101] (5,7]  (7,9]  (3,5]  (5,7]  (3,5]  (5,7]  (5,7]  (3,5]  (3,5]  (5,7]
#> [111] (5,7]  (5,7]  (7,9]  (7,9]  (7,9]  (5,7]  (7,9]  (5,7]  (5,7]  (5,7]
#> [121] (5,7]  (5,7]  (3,5]  (3,5]  (5,7]  (5,7]  (3,5]  (7,9]  (5,7]  (5,7]
#> [131] (3,5]  (7,9]  (5,7]  (5,7]  (5,7]  (5,7]  (5,7]  [1,3]  (3,5]  (5,7]
#> [141] (3,5]  (5,7]  (5,7]  (3,5]  (3,5]  (3,5]  (3,5]  (3,5]  (3,5]  (5,7]
#> [151] (7,9]  (7,9]  (3,5]  (5,7]  (7,9]  [1,3]  (7,9]  (7,9]  (5,7]  (5,7]
#> [161] (7,9]  (5,7]  (3,5]  (5,7]  (3,5]
#> Levels: [1,3] (3,5] (5,7] (7,9] (9,11]
```

The working mechanism of `cut_width()` is that it makes groups of width `width` and create a factor with the levels be the different groups. For example, the first observation has `oa_qual` = 6, belong to the (5,7] group.

Note there are also functions `cut_interval()` and `cut_number()` which also discretise continuous variable into a discrete one by making groups with equal range and equal number of observations, respectively.

Now, you can compare the distributions of a continuous variable on the constructed groups from another continuous variable.

```
ggplot(data = remove_missing(sahp, vars="oa_qual")) + geom_boxplot(aes(x = cut_width(oa_qual, wid
```

bookdown-demo_files/figure-latex/unnamed-chunk-298-1.pdf This agrees perfectly with our intuition that houses with higher overall quality have higher sale prices.

3.14.4 Map aesthetics to boxplot

Before talking about aesthetics, let's create a boxplot of `sale_price` for different values of `house_style`.

```
ggplot(data = na.omit(sahp), aes(x = house_style, y = sale_price)) + geom_boxplot() + scale_x_dis
```

bookdown-demo_files/figure-latex/unnamed-chunk-299-1.pdf

Note that we only show the two boxplots with `house_style` equaling `"1Story"` or `"2Story"`. To simply the codes, it is sometimes helpful to store the intermediate plot object and build additional plots on top of it. For example, we can generate the same boxplot using the following two steps.

```
g <- ggplot(data = na.omit(sahp), aes(x = house_style, y = sale_price)) + scale_x_discrete(limits
g + geom_boxplot()
```

bookdown-demo_files/figure-latex/unnamed-chunk-300-1.pdf

a. map the grouping variable to `color`

First, let's try to map the variable `house_style` to the `color` aesthetic.

```
g + geom_boxplot(mapping=aes(color = house_style))
```

bookdown-demo_files/figure-latex/unnamed-chunk-301-1.pdf

We can see that the boxplots have different colors according to the value of `house_style`.

b. map the grouping variable to `fill`

You can also use the `fill` aesthetic to fill in the boxes with different colors according to the value of `house_style`.

```
g + geom_boxplot(mapping=aes(fill = house_style))
```

bookdown-demo_files/figure-latex/unnamed-chunk-302-1.pdf

c. map a third variable to `color`

So far, we have only mapped the discrete variable on the x-axis to the aesthetic. You can map a third variable to an aesthetic if a further refined comparision is needed. Let's try to map the `oa_qual > 5` to `color`.

```
g + geom_boxplot(mapping=aes(color = oa_qual > 5))
```

bookdown-demo_files/figure-latex/unnamed-chunk-303-1.pdf

You will get a boxplot for each combination of `house_style` and `oa_qual` grouped by the variable `house_style`, just like when we create the bar charts in Section 3.10.

As before, you can also cut a continuous variable and map it to aesthetic.

```
g + geom_boxplot(mapping=aes(color = cut_width(oa_qual, 2)))
```

bookdown-demo_files/figure-latex/unnamed-chunk-304-1.pdf

d. map a third variable to `fill`

Similarly, you can also map the variable to the `fill` aesthetic.

```
g + geom_boxplot(mapping=aes(fill = oa_qual > 5))
```

bookdown-demo_files/figure-latex/unnamed-chunk-305-1.pdf

e. global aesthetics

In addition to mapping variables to aesthetics, you can also use global aesthetics in boxplot. For example, to make the box green and the lines and points red, you can use

```
g + geom_boxplot(fill = "green", color = "red")
```

bookdown-demo_files/figure-latex/unnamed-chunk-306-1.pdf

If you want to change the shape and size of the outliers, you can set the arguments `outlier.shape` and `outlier.size`.

```
g + geom_boxplot(outlier.color = "green", outlier.shape = 2, outlier.size = 3)
```

bookdown-demo_files/figure-latex/unnamed-chunk-307-1.pdf

3.14.5 Notched Boxplots

In addition to the regular boxplot, there is a more sophiscated version, called **notched boxplot**. We can generate such a boxplot by setting the global aesthetic `notch = TRUE` in the `geom_boxplot()` function.

```
ggplot(data = sahp) + geom_boxplot(aes(x = "", y = sale_price), notch = TRUE)
```

bookdown-demo_files/figure-latex/unnamed-chunk-308-1.pdf

In a notched box plot, a notch is generated around the median, with the vertical width on each side being 1.58 times IQR divided by the squared root of the sample size: $1.58 * IQR/sqrt(n)$. This gives a roughly 95% confidence interval for the median. As a result, if the notches of two boxplots do not overlap, it offers evidence of a statistically significant difference between the two medians. In this example, the upper and lower points of the notch are

```
median(sale_price) + 1.58*IQR(sale_price)/sqrt(length(sale_price))
```

```
#> [1] 166.7854
```

```
median(sale_price) - 1.58*IQR(sale_price)/sqrt(length(sale_price))
```

```
#> [1] 149.1146
```

3.15 Violin Plots

In this section, we introduce how to combine music with R via creating **violin plots**. In fact, the violin plot elegantly combines boxplot (Section 3.14) and density plots (Section 3.13) into a single plot.

3.15.1 The basic violin plot

Let's say we want to generate a basic violin plot for the variable `sale_price` in the `sahp` dataset.

```
library(r02pro)
library(ggplot2)
ggplot(data = sahp, aes(x = "", y = sale_price)) + geom_violin()
```

bookdown-demo_files/figure-latex/unnamed-chunk-310-1.pdf

To introduce the detail of the generation process of violin plot, it is helpful to review the density plot.

```
ggplot(data = sahp, aes(y = sale_price)) + geom_density()
```

bookdown-demo_files/figure-latex/unnamed-chunk-311-1.pdf

Looking at these two plots, it is easy to see that the basic violin plot is nothing but a Mirrored Density plot with the kernel density estimates on each side.

3.15.2 Violin plot with boxplot

Usually, the violin plot includes the boxplot inside it, providing extra information about the data. To do this, we just add the boxplot layer on top of the violin plot.

```r
ggplot(data = sahp, aes(x = "", y = sale_price)) +
  geom_violin() +
  geom_boxplot(width = 0.1)
```

bookdown-demo_files/figure-latex/unnamed-chunk-312-1.pdf

Here, we set the aesthetic `width = 0.1` in the boxplot to make it thinner.

Just like in the boxplot, we can compare the distributions of a continuous variable for different values of a discrete variable. We can achieve this by mapping the discrete variable to the x axis.

```r
ggplot(data = sahp, aes(x = house_style, y = sale_price)) +
  geom_violin() +
  geom_boxplot(width = 0.1)
```

bookdown-demo_files/figure-latex/unnamed-chunk-313-1.pdf

We can restrict the x-axis to a subset of the possible `house_style` values.

```r
ggplot(data = sahp, aes(x = house_style, y = sale_price, color = house_style)) +
  geom_violin() +
  geom_boxplot(width = 0.1) +
  scale_x_discrete(limits=c("1Story", "2Story"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-314-1.pdf

Similarly, we can map a third variable to an aesthetic.

```
ggplot(data = remove_missing(sahp, vars="oa_qual"), aes(x = house_style, y = sale_price, fill = c
  geom_violin() +
  geom_boxplot(width = 0.1) +
  scale_x_discrete(limits=c("1Story", "2Story"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-315-1.pdf

As you can see, the boxplot doesn't align well inside the violin plot. To fix this issue, you can add the global aesthetic `position = position_dodge(0.9)` to both geoms.

```
ggplot(data = remove_missing(sahp, vars="oa_qual"), aes(x = house_style, y = sale_price, fill = c
  scale_x_discrete(limits=c("1Story", "2Story")) +
  geom_violin(position = position_dodge(0.9)) +
  geom_boxplot(width = 0.1, position = position_dodge(0.9))
```

bookdown-demo_files/figure-latex/unnamed-chunk-316-1.pdf

You can also try to add other global aesthetics to both geoms to change their appearances.

```
ggplot(data = sahp, aes(x = house_style, y = sale_price, fill = house_style)) +
  geom_violin(color = "green", size = 2) +
  geom_boxplot(width = 0.1, color = "blue", size = 1) +
  scale_x_discrete(limits=c("1Story", "2Story"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-317-1.pdf

3.16 Error Bars

So far, we have learned several ways to compare continuous distributions for different groups, including the density plot, boxplot, and the violin plot. In this lesson, we introduce another visualization method, called **error bar**, which is a graphical representations of the uncertainty in a certain measurement. Let's prepare a ggplot object to get started.

```
library(r02pro)
library(ggplot2)
g <- ggplot(data = sahp, aes(x = house_style, y = sale_price)) +
  scale_x_discrete(limits=c("1Story", "2Story"))
```

Here, we visualize the error bar as the mean ± standard error.

To show the mean of `sale_price` for different `house_style` groups, you can use the `geom_point()` function with arguments `stat = "summary"` and `fun = "mean"`.

```
g  + geom_point(stat = "summary", fun = "mean")
```

bookdown-demo_files/figure-latex/unnamed-chunk-319-1.pdf

To add error bars to the mean points, we use `geom_errorbar()` coupled with the `fun.data = "mean_se"` argument.

```
g  + geom_point(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se")
```

bookdown-demo_files/figure-latex/unnamed-chunk-320-1.pdf

You can control the width of the errorbar by setting the `width` argument.

```
g  + geom_point(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.2)
```

bookdown-demo_files/figure-latex/unnamed-chunk-321-1.pdf

This plot is called a **mean-error-bar** plot. Let's take a detailed look at this plot. The distance from the mean point to the lower bound of the error bar is the so called **standard error**, which is defined as the standard deviation of the sample divided by the square root of the sample size. Let's try to calculate the upper and lower bounds of the error bar.

```r
sale_price_1_story <- sahp$sale_price[sahp$house_style == "1Story"]
sale_price_1_story_se <- sd(sale_price_1_story)/sqrt(length(sale_price_1_story))
mean(sale_price_1_story) + sale_price_1_story_se
```

```
#> [1] 192.251
```

```r
mean(sale_price_1_story) - sale_price_1_story_se
```

```
#> [1] 173.8044
```

You may notice that we are referring the data as sample since the `sahp` dataset contains only a subset of the population with all houses in the world. An important application of error bar is to construct the 95% confidence interval of the population mean. You know that the 95% confidence interval for a normal distribution is mean ± 1.96*se. If the data follows a normal distribution, you can construct such a confidence interval by setting the multiplier of the standard error in the error bar to be 1.96.

```r
g  + geom_point(stat = "summary", fun = "mean") +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.2, fun.args = list(mult = 1.96)
```

bookdown-demo_files/figure-latex/unnamed-chunk-323-1.pdf

Of course, you can also add the error bar on top of the other plots like the boxplot or violin plot.

```r
g  + geom_point(stat = "summary", fun = "mean") +
  geom_boxplot() +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.2)
```

bookdown-demo_files/figure-latex/unnamed-chunk-324-1.pdf

```
g  + geom_point(stat = "summary", fun = "mean") +
  geom_violin() +
  geom_errorbar(stat = "summary", fun.data = "mean_se", width = 0.2)
```

3.17 Arrange Multiple Plots

So far, you have mastered many geoms, and you know one popular way to add the information of additional variables to a plot is mapping them to certain aesthetics. Another way to achieve this goal is to divide the data into different groups according to the additional variables, generate different plots for each group, and arrange the plots into **facets**. This is particularly useful for categorical variables.

Let's use the `sahp` dataset and generate a scatterplot for `sale_price` vs. `liv_area` and save it as `g`.

```
library(ggplot2)
library(r02pro)
g <- ggplot(data = na.omit(sahp)) + geom_point(mapping = aes(x = liv_area, y = sale_pr
```

3.17.1 Facet wrap

The first function is called `facet_wrap()`, which generate separate plots for each category and wrap the plots into 2d panels.

Let's first divide to data according to `oa_qual`.

```
g + facet_wrap("oa_qual")
```

bookdown-demo_files/figure-latex/unnamed-chunk-327-1.pdf

You can see that there are nine different scatterplots between `sale_price` and `liv_area` according to the value of `oa_qual`. The scatterplots are nicely arranged in a 3-by-3 grid with the value of `oa_qual` on top of each subplot.

Note that the scales of all nine plots are the same, which make it straightforward to compare between different groups. On the other hand, the points for some of the groups (e.g., `oa_qual == 5`) are scrambled together, making us difficult to see the details. To make the subplots having their own scales, you can set `scales = "free"`.

```
g + facet_wrap("oa_qual", scales = "free")
```

bookdown-demo_files/figure-latex/unnamed-chunk-328-1.pdf

From the plot, it is clear that each subplot has its own scale depending on the data range in the subset of the data. If you want to make the subplots to have fixed scale only on the x-axis or the y-axis, you can set `scales = "free_y"` or `scales = "free_x"` respectively.

In addition to the default layout, you can set the desired number of rows or columns. For example, to arrange the plots in two rows for the variable `kit_qual`, you can use

```
g + facet_wrap("kit_qual", nrow = 2)
```

bookdown-demo_files/figure-latex/unnamed-chunk-329-1.pdf

In addition to using one variable to form the subgroups, you can also use multiple variables by using a vector of their names in the `facet_wrap()` function.

```
g + facet_wrap(c("kit_qual","central_air"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-330-1.pdf

This will show the plots for combinations of `kit_qual` and `central_air`. For example, the top left plot is the scatterplot between `liv_area` and `sale_price` for houses with `kit_qual == "Average"` and `central_air == "Y"`. Let's find out how many different values are there for `kit_qual` and `central_air`.

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

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

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

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

Clearly, there are in total eight possible combinations of values for these two variables. Upon a careful look, you may realize there is no plot for `kit_qual == "Excellent"` and `central_air == "N"`. The reason is due to the fact there are no houses satisfying both criteria. Let's verify as follows.

```
sum(sahp$kit_qual == "Excellent" & sahp$central_air == "N")
```

```
#> [1] 0
```

Note that the `facet_wrap()` function can be combined with any geoms we have learned. Let's see an example of bar charts.

```
a<- ggplot(data = sahp) + geom_bar(mapping = aes(x = kit_qual))
a + facet_wrap("house_style")
```

bookdown-demo_files/figure-latex/unnamed-chunk-333-1.pdf

3.17.2 Facet grid

```r
g + facet_grid(rows = vars(kit_qual), cols = vars(central_air))
```

bookdown-demo_files/figure-latex/unnamed-chunk-334-1.pdf

In addition to facet wrap, you can also use the function `facet_grid()` to form a matrix of plots defined by row and column faceting variables. It is mostly useful when you have two discrete variables, and most combinations of the variables exist in the data.

Comparing with the plot generated by `facet_wrap()`, `facet_grid()` will also show empty plots if there are no observations with certain combinations, e.g. `kit_qual == "Excellent"` and `central_air == "N"`. This particular way to arrange plots is very informative.

Similar to `facet_wrap()`, you can also allow the scales of each subplot to be different by setting the `scales` argument.

```r
g + facet_grid(rows = vars(kit_qual), cols = vars(central_air), scales = "free")
```

bookdown-demo_files/figure-latex/unnamed-chunk-335-1.pdf

3.18 Customization of Labels and Titles

In Section 3.11, we discussed the customization of the x and y axes in ggplot. Now, we will introduce other types of customization, and the concept of themes.

Let's first review the scatterplot we created between `liv_area` and `sale_price` and save it into as an object `g`.

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

a. customize x and y labels and title

By default, the x and y labels are the variables names, and there is no title for the plot. You can customize the x and y labels using the `xlab()` and `ylab()` function, and add a title with the `ggtitle()` function.

```
g1 <- g + xlab("Living Area") +
  ylab("Sale Price") +
  ggtitle("Price vs. Area")
g1
```

bookdown-demo_files/figure-latex/unnamed-chunk-337-1.pdf

b. customize the font of the x and y breaks

In addition, you can further customize the font of the x and y breaks using the `theme()` function with the argument `axis.text`.

```
g1 + theme(axis.text = element_text(size = 25, color = "red"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-338-1.pdf

c. customize the font of labels

To customize the font, you can use `axis.title` argument to change the size, color, and face of the labels.

```
g1 + theme(axis.title = element_text(size = 18, color = "red", face = "italic"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-339-1.pdf

d. customize the font of the title

Similarly, you can use the `plot.title` argument to customize the font of the title.

```
g1 + theme(plot.title = element_text(size = 24, color = "magenta", face = "bold"))
```

bookdown-demo_files/figure-latex/unnamed-chunk-340-1.pdf

e. center the title

Sometimes, we may want to center the title. We can achieve this by setting the `hjust` parameter.

```
g1 + theme(plot.title = element_text(size = 24, color = "magenta", face = "bold",hjust = 0.5))
```

bookdown-demo_files/figure-latex/unnamed-chunk-341-1.pdf

f. mix

Apparently, you are free to mix all the different customizations. Let's see an example as below.

```
g1 + theme(axis.title = element_text(size = 18, color = "red", face = "italic"), axis.text = elem
```

bookdown-demo_files/figure-latex/unnamed-chunk-342-1.pdf

g. save as a theme

As you can see from the code, the code gets complicated if we want to customize many things at the same time. To save time, you can actually save the desired into an R object and reuse it later.

```
mytheme <- theme(axis.title = element_text(size = 18, color = "red", face = "italic"), axis.text
```

Then, we can generate the same plot with `mytheme` using

```
g1 + mytheme
```

Similarly, you can add another layer of smoothline fit and apply `mytheme`.

```
g1 + geom_smooth() + mytheme
```

You may notice that title is not accurate. So, we want to change the title to reflect the addition of the smoothline fit. Let's simply add another `ggtitle()` function to overwrite the existing one in the theme.

```
g1 + geom_smooth() + mytheme +
  ggtitle("Scatterplot + smoothline")
```

3.19 Summary of Geoms & Grammatical Structure of `ggplot()`

Today we will have a review lesson for using `ggplot()` to do data visualization. First, let's review all the geoms we have covered.

Code	Name	Section
‘dnorm(x, mean, sd)‘	probability density function	\@ref(pdf)
‘pnorm(q, mean, sd)‘	cumulative distribution function	\@ref(cdf)
‘qnorm(p, mean, sd)‘	quantile function	\@ref(qf)
‘rnorm(n, mean, sd)‘	random number generator	\@ref(rng)

Format	Tag example
Headings	=heading1===heading2=====heading3===
New paragraph	A blank line starts a new paragraph
Source code block	// all on one line {{{ if (foo) bar else baz }}}

- One continuous variable (e.g. `sale_price`): `geom_histogram(aes(x = sale_price))`, `geom_density(aes(x = sale_price))` `geom_boxplot(aes(x = "", y = sale_price))`, `geom_violin(aes(x = sale_price))`, `geom_errorbar(aes(x = "", y = sale_price), stat = "summary", fun.data = "mean_se")`
- One discrete variable (e.g. `kit_qual`): `geom_bar(mapping = aes(x = kit_qual))`
- Two continuous variables (e.g. `liv_area` and the `sale_price`): `geom_point(mapping = aes(x = liv_area, y = sale_price))`, `geom_smooth(mapping = aes(x = liv_area, y = sale_price))`
- Two discrete variables (e.g. `kit_qual` and `heat_qual`): `geom_jitter(mapping = aes(x = kit_qual, y = heat_qual))`, `geom_count(mapping =`

```
  aes(x = kit_qual, y = heat_qual))
```

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 Handing 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 `NA`s 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`.

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)

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.

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 .xlxs.

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_x
```

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
#>   <dttm>                 <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 <chr>, sale_price <dbl>, good_qual <lgl>
```

If we only want to import a portion of the data, let' 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
#>   <dttm>                <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. ‘TRUI
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.

```r
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.

```r
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.

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

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

```r
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.

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

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

```r
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
```

```r
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.

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

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

```r
dig_num <- 826
dig_num
```

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

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

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

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

```r
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 simplier 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.

```r
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.

```r
?ahp
```

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

```r
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.

```r
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 orginal 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 μ and variance σ^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
‘dnorm(x, mean, sd)‘	probability density function	\@ref(pdf)
‘pnorm(q, mean, sd)‘	cumulative distribution function	\@ref(cdf)
‘qnorm(p, mean, sd)‘	quantile function	\@ref(qf)
‘rnorm(n, mean, sd)‘	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.

```r
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.

```r
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.

```r
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.

```r
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`.

```r
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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-411-1.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 = 0.5))
norm_dat <- rbind(norm_dat_1, norm_dat_2, norm_dat_3)
ggplot(norm_dat) + geom_line(aes(x = x, y = pdf, color = dist))
```

bookdown-demo_files/figure-latex/unnamed-chunk-412-1.pdf

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

$$F(x) = P(X \le x).$$

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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-414-1.pdf

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))
```

bookdown-demo_files/figure-latex/unnamed-chunk-417-1.pdf

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 $N(0, 1)$.

```r
rnorm(3, mean = 0, sd = 1) #generate 3 random numbers from N(0, 1)
rnorm(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.

```r
set.seed(724)
rnorm(3)
```

Now, let's run it one more time.

```r
set.seed(724)
rnorm(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 `rnorm()`, you will get totally different results. See the following example.

```r
set.seed(724)
rnorm(1)
rnorm(3)
```

By setting a different random seed, you will see different results as the following example.

```r
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 specify the targeted number of elements. By default, the `sample` function take 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 the popu
```

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 samples 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.

```r
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 $Cov(X, Y) = E[(X - E(X))(Y - E(Y))]$.

```r
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.

```r
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.".