Chapter 6: Programming in R

Mark Andrews

Contents

Introduction

When being introduced initially, R is often described as a programming language. This statement is technically correct, and practically very important too, but it is somewhat misleading too and so ought to be put in context. While R does have all the major programming features that we expect in any programming language — functions, conditionals, iterations, etc — and these are the reasons for R's power and extensibility, whenever we use R, especially at the beginning, we are not necessarily programming in the sense as we would be if we were using C/C++, Java, etc. Certainly, whenever we use R, we are writing code whenever and we usually put this code in scripts, which can then be executed as a batch. However, writing loops, functions, custom R

objects classes, etc., is often not done until users reach a certain level of proficiency and confidence. In fact one could use R productively without engaging in programming of this kind at all. This point is meant to reassure newcomers to R that learning to use R productively does not require the same considerable initial investment that might be required when learning general purpose programming languages. We can start with learning individual R commands, build up a repertoire of widely used commands, and eventually after a certain level of familiarity and confidence has been achieved, we can start learning how to program in R.

In this chapter, we aim to provide an introduction to some of the major programming features of R. We'll begin with *functions* both because they can be very simple to use and because of the major role functions play in programming in R generally. We will then consider *conditionals*, which allow us to execute different blocks of code depending on whether certain conditions are true. We will then consider *iterations*, also known as *loops*. This will lead on to *functionals*, which can often take the place of traditional loops in R. As part of our consideration of functionals, we will also consider the `purrr` package that is part of the tidyverse. The aim of `purrr` is to make functionals easier and more efficient to use.

There is more to programming in R than what we cover here. For example, object oriented programming and non-standard evaluation are both practically very important features of R programming. To keep our coverage brief and introductory, we will not cover either here, but highly recommend the book by Wickham (2019) for coverage of these and other R programming topics.

Functions

Functions in R, just like in any other programming language, allow us to create custom commands to perform specific calculations or carry out specific tasks. Whenever we find ourselves repeatedly using identical or similar code statements, we can create a function to execute this code. In R, functions usually, but not necessarily, take some R objects as input and always returns a new object (though this could be the `NULL` object, which signifies the absence of a defined value). We can use functions to carry out any calculations or any procedure that we could perform using any other R code.

Let's start with a numerical calculation. Let's say we have a vector of probabilities, and we want to calculate the logarithm, to a specified base, of the odds of these probabilities. In other words, let's say we have a vector of probabilities

$$p_1, p_2 \dots p_n$$

and we want to calculate

$$\log_b \left(\frac{p_i}{1 - p_i} \right) \quad \text{for } i \in 1 \dots n.$$

This calculation is simple to perform using R. For example, let's say our probabilities are as follows.

```
p <- c(0.1, 0.25, 0.5, 0.75)
```

Then, the logarithm, to base 2, of the odds of the values in `p` are calculated as follows.

```
log(p/(1-p), base = 2)
#> [1] -3.169925 -1.584963  0.000000  1.584963
```

If, on the other hand, we wanted to logarithm to base 3 or base 5 of these odds, we could do the following.

```
log(p/(1-p), base = 3)
#> [1] -2 -1  0  1
log(p/(1-p), base = 5)
#> [1] -1.3652124 -0.6826062  0.0000000  0.6826062
```

Likewise, if we had the another vector of probabilities we could calculate the log to base 2 of their odds of these values as follows.

```
q <- c(0.33, 0.67, 0.99)
log(q/(1-q), base = 2)
#> [1] -1.021695  1.021695  6.629357
```

Clearly, in these examples, we are repeating the same or similar code statements. In general, we should avoid doing this. Not only is it tedious to repeatedly type the same code, but every time we do so, we introduce the possibility, however small, of a coding error. We can therefore create a function that takes any vector of probabilities and returns the logarithm of their odds to any desired base. We can define this function as follows:

```
log_odds <- function(p, b){
  log(p/(1-p), base = b)
}
```

To understand what is being defined here, first note that all functions in R are defined by the `function` keyword. The variable names within the brackets after the `function` statement are the function's so-called input *arguments*. In this case, we have two arguments, named `p` and `b`. If the function takes no input, then we'd simply write `function()` here. The code within the `{}` is known the function's *body*. This is otherwise normal R code, but as we will see, it is running in an informationally encapsulated environment. In this example, this code is operating on two variables, `p` and `b`, and these variables are what are we will supply as inputs whenever we call the function. The value that this code in the body calculates is then what the function returns. In general, as we will discuss below, the value of the last expression in the function's body is the value that it returns. In this case, there is obviously just one expression in the body, and the value of this expression is what is returned. In order to use a function in R, we usually (though not necessarily, as we will see) assign it to some name using the usual assignment operator. In this case, we assign the function to the name `log_odds`. We can now *call* this function to the run the statements above as follows.

```
log_odds(p, 2)
#> [1] -3.169925 -1.584963  0.000000  1.584963
log_odds(p, 3)
#> [1] -2 -1  0  1
log_odds(p, 5)
#> [1] -1.3652124 -0.6826062  0.0000000  0.6826062
log_odds(q, 2)
#> [1] -1.021695  1.021695  6.629357
```

Note that although the body of the `log_odds` function lies within the `{}` curly braces, when a function's body contains just a single expression, we may omit the braces. For example, we could write `log_odds` as follows.

```
log_odds <- function(p, b) log(p/(1-p), base = b)
```

The curly braces can always be used, no matter how simple the function, but in particularly simple cases, it is not uncommon for them to be omitted.

Input arguments

Notice that in the function definition, we stated that it takes two input arguments `p` and `b`, and that the code in the body explicitly operates on `p` and `b`. As the above examples make clear, the names we use for the input argument in the function *definition* are arbitrary and do not have to correspond to the names of the variables that we use when we *call* the function. By default, whatever variable is passed in first is what the function internally defines as `p` and the second variable is what is defines as `b`. For example, consider the following code.

```
probs <- c(0.25, 0.75, 0.9)
log_base <- 2
log_odds(probs, log_base)
#> [1] -1.584963  1.584963  3.169925
```

The function `log_odds` takes the vector `probs` and the number `log_base` and internally refers to them as `p` and `b`, respectively, and then calculates and returns `log(p/(1-p), base = b)`. As we see in the following code, we may also explicitly indicate which variables are mapped to `p` and `b`.

```
log_odds(p=probs, b=log_base)
#> [1] -1.584963  1.584963  3.169925
```

When using explicit assignment like this, the order of the arguments no longer matters. Thus, we can write the above code as follows.

```
log_odds(b=log_base, p=probs)
#> [1] -1.584963  1.584963  3.169925
```

Default values for arguments

In the function we defined above, we had to explicitly provide both the probabilities and the base of the logarithms as input arguments. In some cases, however, we may prefer to allow some input arguments to take default values. For example, in this case, we may prefer the base of the logarithms to be 10 by default. We define default values in the function definition, as in the following example.

```
log_odds2 <- function(p, b=10){
   log(p/(1-p), base = b)
}
```

We can use this function exactly as we did with the original version, i.e. by providing two input arguments explicitly, as in the following examples.

```
log_odds2(probs, log_base)
#> [1] -1.584963  1.584963  3.169925
log_odds2(b = 3, p = probs)
#> [1] -1  1  2
```

However, if we do not include the `b` argument explicitly, then it will default to `b=10`, as in the following example.

```
log_odds2(probs) # here b=10
#> [1] -0.4771213  0.4771213  0.9542425
```

Optional arguments

A function can also have an optional arguments indicated by `...` in the arguments list in the function definition. This is often to pass arguments to functions that are called within functions. Consider the following function that calculates a polynomial function.

```
f_poly <- function(x, y, s, t){
  x^s + y^t
}
```

Let's say that we want to create a function that returns the logarithm to some specified base of `f_poly` for any given values of `x`, `y`, `s`, and `t`. We could do the following.

```
log_f_poly <- function(b=2, x, y, s, t){
  log(f_poly(x, y, s, t), base = b)
}
```

However, an easier option would be the following.

```
log_f_poly <- function(..., b=2){
  log(f_poly(...), base = b)
}
```

This will capture all the arguments, other than `b`, in the call of `log_f_poly` and pass them to `f_poly`. We can see this in action in the following example.

```
x <- c(0.5, 1.0)
y <- c(1.0, 2.0)
```

```
s <- 2
t <- 3
log_f_poly(x, y, s, t, b=2)
#> [1] 0.3219281 3.1699250
```

Note that we may obtain the optional arguments as a list given by `...` by using `list(...)` in the code body, as in the following trivial example function, where we return the optional argument list.

```
f <- function(...){
  list(...)
}
f(x=1, y=2, z=3)
#> $x
#> [1] 1
#>
#> $y
#> [1] 2
#>
#> $z
#> [1] 3
```

By using `list(...)`, we can always extract and operate upon all the information provided by optional arguments.

Missing arguments

Consider the following simple function.

```
add_xy <- function(x, y, z){
  x + y
}
```

The function definition states that there will be three input arguments, `x`, `y`, and `z`. However, in this case, it can be used without error if we only supply `x` and `y` because `z` is not used in the code body.

```
add_xy(5, 8)
#> [1] 13
```

In this case, we say that the `z`, which is explicitly stated as an input argument, is a missing argument. Clearly, missing arguments will not necessarily raise an error but we can always test whether any given argument is missing by using the `missing` function within the code body. In the following function, we test whether each of the three input arguments are missing or not.

```
is_missing_xyz <- function(x, y, z){
  c(missing(x),
    missing(y),
    missing(z))
}

is_missing_xyz(1, 2)
#> [1] FALSE FALSE  TRUE
is_missing_xyz(z = 1, y = 42)
#> [1]  TRUE FALSE FALSE
is_missing_xyz(5, 4, 3)
#> [1] FALSE FALSE FALSE
```

As we will see in examples later throughout this book, the ability to test for missing inputs provides additional flexibility in how we can use functions.

Function return values

The functions we have defined thus far have had single expressions in their bodies. The values of these expressions are what are returned by the functions. Functions, however, can have arbitrarily many statements and expressions in their body. When there are multiple statements, the value of the *last* expression is the value that is returned. To illustrate this, we can make a multi-statement version of `log_odds2` as follows.

```
log_odds3 <- function(p, b=10){
  odds <- p / (1 - p)
  log(odds, base=b)
}
log_odds3(probs)
#> [1] -0.4771213  0.4771213  0.9542425
```

A variant of the function `log_odds3` is the following.

```
log_odds4 <- function(p, b=10){
  odds <- p / (1 - p)
  log_of_odds <- log(odds, base=b)
  log_of_odds
}
log_odds4(probs)
#> [1] -0.4771213  0.4771213  0.9542425
```

In both `log_odds3` and `log_odds4`, there are multiple code statements in their bodies. In both cases, the value returned is the value of the final expression in the body. It is also possible to have an explicit `return` statement in the function's code body.

```
log_odds5 <- function(p, b=10){
  odds <- p / (1 - p)
  log_of_odds <- log(odds, base=b)
  return(log_of_odds)
}
log_odds5(probs)
#> [1] -0.4771213  0.4771213  0.9542425
```

If a `return` statement is used, whenever it is reached, the function will immediately return its value and the remainder of the code in the body, if any, is not executed. In other words, the `return` statement allows us to *break out* of a function early, which is a useful feature that we will see below after we consider conditionals. It is conventional in R to only use explicit `return` statements for this break out purpose, and to normally just use the final expression in the body as the function's returned value.

Function scope and environment

Consider the following function.

```
assign_x <- function(){
  x <- 17
  x
}
```

This function will takes nothing as input, and in the body assigns the value of 17 to the variable `x` and then returns this value. We can use it as follows.

```
x <- 42
assign_x() # returns 17
#> [1] 17
x          # but x is still 42
#> [1] 42
```

We see that the original value of `x` remains unchanged despite the fact that we have the statement `x <- 17` within the function's body. This is because the variable the `x` within the function's body is a *local* variable, or rather it is a variable defined within a *local environment* belonging to the function. Variables in the local environment are not visible outside of that environment, which in all the examples thus far is the *global environment*, which is the top level environment in the R session. In this sense, any function is informationally encapsulated: the variables defined by normal assignment operations within the function's local environment do not exist outside of the function, nor do normal assignment operations within the function's local environment affect variables outside the function. Here is another example of this phenomenon.

```
assign_x2 <- function(x){
  x[2] <- 42
  x
}

x <- c(2, 4, 8)

# The `x` within `assign_x2` is changed
assign_x2(x)
#> [1]  2 42  8
# The `x` in the global environment is unchanged
x
#> [1] 2 4 8
```

Although variables in the local environment are not visible or usable in the global environment, the opposite is not true. Variables in the global environment can be used in the local environment. Consider the following example.

```
increment_x <- function(){
  x + 1
}

x <- 42
increment_x()
#> [1] 43
```

The body of `increment_x` refers to a variable `x` that is not defined in the body, nor is it passed in as an input argument. When this happens, R looks for `x` outside the local environment. In the example above, it finds it in the global environment with the value of 42. It then increments that value by 1 and returns the result. Even in this case, however, the value of `x` in the global environment remains unchanged.

```
x <- 101
increment_x()
#> [1] 102
x
#> [1] 101
```

Thus far, we have mentioned a function's local environment and contrasted this with the global environment. However, functions may be nested. In that case, we have can multiple levels of environments. Consider the following example.

```
f <- function(){

  x <- 1

  g <- function(){
    x + 1
  }
```

```
  h <- function(){
    y + 2
  }

  c(g(), h())
}
```

In this example, the functions `g` and `h` have their own local environment, but these environments are within the local environment of the function `f`, which is within the global environment. In this case, we say that the environment of `f` is the *parent* environment of `g` and `h`, and the global environment is the parent environment of `f`. This nesting of environments determines how values of variables are looked up. For example, when `g` is called it looks for `x`, which does not exist in its local environment, so it looks for it in its parent environment, which is the local environment of `f`. When `h` is called, it looks for `y`, which exists neither within its own local environment, or within its parent environment, so it must look for it in the global environment, which is its grandparent environment (the parent environment of its parent environment). We can see this function in action in the following example.

```
y <- 42
f()
#> [1]  2 44
```

An important feature of function environments is that they are defined by where the function is defined not where it is called. This becomes important when a function is returned by another function. Consider the following example.

```
f <- function(){
  y <- 42

  function(x){
    x + y
  }

}

g <- f()
```

In this example, `g`, which is the output of the call of `f`, is a function whose parent environment is the environment of `f`. As such, we can do the following.

```
y <- 21
g(17)
#> [1] 59
```

Note that the result here is `17 + 42` and not `17 + 21`. The `g` function is defined as `function(x) x + y`, and so it must look to its parent's, or grandparent's and so on, environment to find the value of `y`. Although `y` takes the value of 21 in the global scope, it takes the value of 42 in the environment of `f`, which is `g`'s parent environment.

Finally, although we mentioned that normal assignments within a local environment do not affect values in the parent, or other ancestor, environments, the special assignment operator `<<-` can be used to assign value in the parent environment. As a simple example, consider the following.

```
f <- function(){
  x <<- 42
}

x <- 17
```

```
f()
x
#> [1] 42
```

In this case, we see that the assignment of the value of 42 to `y` has been applied to the parent environment of `f`, which is the global environment.

Anonymous functions

In all the functions above, the functions were assigned to some name. This is not necessary. Consider the following example.

```
f <- function(x, g){
  g(sum(x))
}
```

Here, `f` takes an object `x` and a function `g` and calls `g(sum(x))` and returns the result. We can pass in any existing R function we wish as the value of `g`, as in the following examples.

```
x <- c(0.1, 1.1, 2.7)
f(x, log10)
#> [1] 0.5910646
f(x, sqrt)
#> [1] 1.974842
f(x, tanh)
#> [1] 0.9991809
```

Of course, we can also pass in any function we have defined ourselves.

```
square <- function(x) x^2
f(x, square)
#> [1] 15.21
```

However, we don't have to assign a name to our custom function and pass in that name. We can instead just pass in the unnamed, or *anonymous*, function itself as in the following examples.

```
f(x, function(x) x^2)
#> [1] 15.21
f(x, function(x) log(x^2))
#> [1] 2.721953
```

Anonymous functions are widely used in R, as we will see when we discuss *functionals* later in this chapter.

We can create a *self executing anonymous function* as in the following example.

```
y <- (function(x, y, z){x + y + z})(1, 2, 3)
y
#> [1] 6
```

Here, a function is created and called immediately and the result assigned to `y`. Given that a function like this can only be invoked once, it may seem pointless. However, it does allow us to write code in an informationally encapsulated environment, where we can possibly re-use variable names from the parent or global environment, and not interfere with variables or add clutter to those environments.

Conditionals

Conditionals allows us to execute some code based on whether some condition is true or not. Consider the following simple example.

```
library(tidyverse)

# Make a data frame
data_df <- tibble(x = rnorm(10),
                  y = rnorm(10))


write_data <- TRUE

if (write_data) {
  write_csv(data_df, 'tmp_data.csv')
}
```

Here, we write `data_df` to a `.csv` file if and only if `write_data` is true. The conditional statement begins with the keyword `if` followed by a round bracketed expression that must contain an expression that evaluates to `TRUE` or `FALSE`. This is then followed by a code block delimited by `{` and `}`. Everything in this code block is executed if and only if the expression is true. Note that if the code block contains only a single expression, just as in the case of the code body of functions, the curly braces surrounding the code block in the conditional can be omitted, as in the following example.

```
if (write_data) write_csv(data_df, 'tmp_data.csv')
```

if . . . else statments

The conditional statements in the examples so far will execute some code if a condition is true, and do nothing otherwise. Sometimes, however, we want to execute one code block if the condition is true and execute an alternative code block if it is false. To do this, we use an *if . . . else* statement, as in the following example.

```
use_new_data <- TRUE

if (use_new_data){
  data_df <- read_csv('data_new.csv')
} else {
  data_df <- read_csv('data_old.csv')
}
```

As we can see, if `use_new_data` is true, we read in the data from `data_new.csv`, and otherwise we read the data in from `data_old.csv`.

Nesting *if* and *if . . . else* statements

We may nest *if* and *if . . . else* statements. In other words, we may evaluate one condition, and if it is true, we may evaluate another condition, and so on. In the following example, if `data_1.csv` exists, we will read in its data. If it does not exist, then we test if `data_2.csv` exists. If it does, we read in its data. If not, we read in the data from `data_3.csv`.

```
if (file.exists('data_1.csv')) {
  data_df <- read_csv('data_1.csv')
} else if (file.exists('data_2.csv')) {
  data_df <- read_csv('data_2.csv')
} else {
  data_df <- read_csv('data_3.csv')
}
```

It should be noted that this example may not seem like nesting of *if . . . else* statements, but rather like a chaining of these statements. However, it is exactly equivalent to the following, clearly nested, *if . . . else* statements.

```
if (file.exists('data_1.csv')) {
  data_df <- read_csv('data_1.csv')
} else {
  if (file.exists('data_2.csv')) {
      data_df <- read_csv('data_2.csv')
  } else {
    data_df <- read_csv('data_3.csv')
  }
}
```

The only difference between these two versions is that the first version omits the (in this case, optional) `{}` after the first occurrence of `else`. As such, whether we see conditional statements like the first version as a chaining or a nesting is not meaningful.

Ultimately, nested (or chained) *if* and *if . . . else* statements allow us to evaluate any binary decision tree. See Figure 1 for some examples.

Nested *if* and *if . . . else* statements can be used when we must choose between $K > 2$ different options. In the following example, we sample $n = 10$ random data points from one of 5 different distributions.

```
n <- 10
distribution <- 'student_t'

if (distribution == 'normal') {
  y <- rnorm(n, mean = 100, sd = 15)
} else if (distribution == 'log_normal') {
  y <- log(rnorm(n, mean = 100, sd = 15))
} else if (distribution == 'student_t') {
  y <- rt(n, df = 10)
} else if (distribution == 'chisq') {
  y <- rchisq(n, df = 3)
} else if (distribution == 'uniform'){
  y <- runif(n, min = -10, 10)
}
```

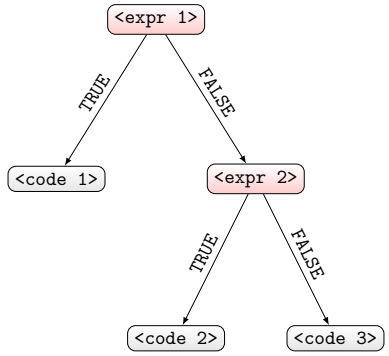
`switch` functions

While, as we've just seen, it is possible to evaluate a non-binary decisions using nested *if* and *if . . . else* statements, sometimes it may be natural and simpler to use a *switch* function, which executes different expressions or code blocks depending on the value of a variable. As an example, we can re-implement the previous example of random number generation using a *switch* function.

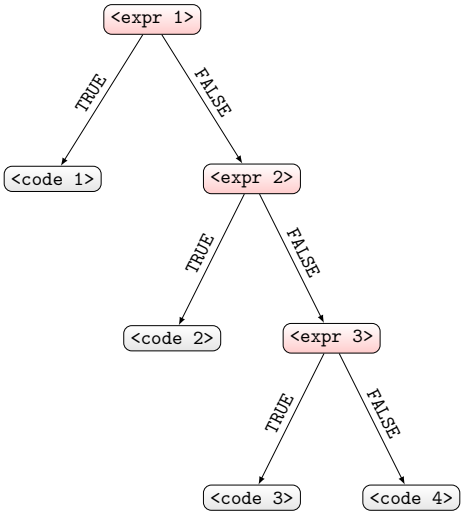
```
distribution <- 'chisq'

y <- switch (distribution,
             normal = rnorm(n, mean = 100, sd = 15),
             log_normal = log(rnorm(n, mean = 100, sd = 15)),
             student_t = rt(n, df = 10),
             chisq = rchisq(n, df = 3),
             uniform = runif(n, min = -10, 10)
)
```

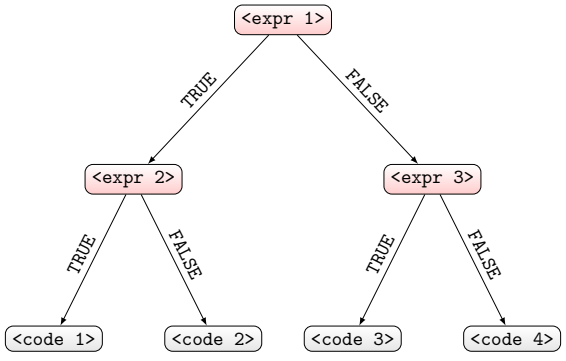
In this example, the `switch` function's first argument is the name of a variable that takes one of five different values, namely `normal`, `log_normal`, `student_t`, `chisq`, or `uniform`. Based on which value is matched, it executes the code that corresponds to that value. In this example, the value of `distribution` is `chisq` and so the `rchisq(n, df = 3)` code is executed.




```
if (<expr 1>) {
  <code 1>
} else if (<expr 2>) {
  <code 2>
} else {
  <code 3>
}
```




```
if (<expr 1>) {
  <code 1>
} else if (<expr 2>) {
  <code 2>
} else if (<expr 3>) {
  <code 3>
} else {
  <code 4>
}
```




```
if (<expr 1>) {
  if (<expr 2>) {
    <code 1>
  } else {
    <code 2>
  }
 } else if (<expr 3>) {
    <code 3>
 } else {
    <code 4>
 }
```

Figure 1: Using nested *if* or *if ... else* statements, we can create any binary decision tree where each terminal node, or leaf, of the tree is a code block and every nonterminal node is an logical expression that evaluates to true or false.

The *switch* function can also be used by choosing the code based on an index rather than by matching a name. In the following example, we choose the second of the five listed options by setting `distribution <- 2`.

```
distribution <- 2

y <- switch (distribution,
             rnorm(n, mean = 100, sd = 15),
             log(rnorm(n, mean = 100, sd = 15)),
             rt(n, df = 10),
             rchisq(n, df = 3),
             runif(n, min = -10, 10)
)
```

In the above examples, the code that is executed is always a simple expression. However, it is possible to have an arbitrary code block instead, as in the following example.

```
distribution <- 'normal'

switch (distribution,
        'normal' = {
          mu <- runif(1, min=-10, max=10)
          sigma <- runif(1, min=0.01, max = 10)
          y <- rnorm(n, mean = mu, sd = sigma)
        },
        'student_t' = {
          mu <- runif(1, min=10, max=20)
          sigma <- runif(1, min=1.01, max = 3)
          y <- mu + rt(n, df=1) * sigma
        }
)
```

In this case, the entire code block corresponding to matched name of `distribution` is executed in the global namespace.

ifelse, if_else, and case_when

If the code being executed by an *if . . . else* statement is simple, such as a single expression, and optionally if we need to have the conditional vectorized, then we can use a `ifelse` function. In the following example, for each value of `reaction_time`, if it is less than 300, we return `'fast'` and otherwise, we return `'slow'`.

```
reaction_time <- c(1000, 300, 200, 250, 450, 300, 250, NA)
ifelse(reaction_time < 300, 'fast', 'slow')
#> [1] "slow" "slow" "fast" "fast" "slow" "slow" "fast" NA
```

As can be seen, we obtain a vector of the same length as `reaction` with values `'fast'` and `'slow'`.

The same functionality of `ifelse` can be obtained with the `if_else` function in `dplyr`.

```
library(dplyr)
if_else(reaction_time < 300, 'fast', 'slow')
#> [1] "slow" "slow" "fast" "fast" "slow" "slow" "fast" NA
```

The `if_else` is identical to `ifelse` but it requires that the expressions corresponding to the true and false values of the logical condition are of the same type. In addition, `if_else` provides us the option of replacing missing values with values of our choice.

The `dplyr` packages also provides the `case_when` function that can be seen as a vectorized version of nested *if . . . else* statements. It is primarily intended to be used as part of `dplyr` pipelines, especially within `mutate` functions, and we also described it in Chapter 3. As an example, we can use `case_when` to create a nonlinear

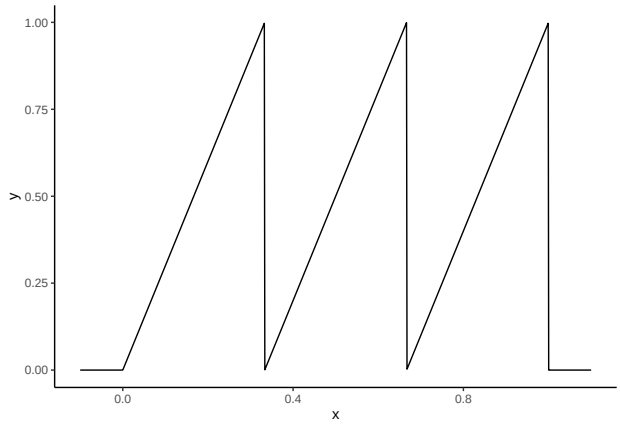


Figure 2: The ternary sawtooth map. This nonlinear function can be easily implemented using a case_when function.

function known as the ternary sawtooth map, which is depicted in Figure 2. First, consider how we could implement this function using nested *if ... else* statements.

```
sawtooth <- function(x){
  if (x < 0){
    0
  } else if (x < 1/3) {
    3 * x
  } else if (x < 2/3) {
    3 * x - 1
  } else if (x <= 1) {
    3 * x - 2
  } else {
    0
  }
}
```

Using `case_when`, this function could be reimplemented as follows.

```
sawtooth2 <- function(x){
  case_when(
    x < 0 ~ 0,
    x < 1/3 ~ 3 * x,
    x < 2/3 ~ 3 * x - 1,
    x <= 1 ~ 3 * x - 2,
    TRUE ~ 0
  )
}
```

Each line within `case_when` (or more precisely, each comma delimited argument), contains a *formula*, which is a statement with a `~`. To the left of `~` is a logical expression, and to the right is the code that is executed if the logical expression is true. Sometimes more than one logical expression within `case_when` is true. That is the case here. For example, if `x < 0` is true, then `x < 1/3` must be true too, and so on. The `case_when` function proceeds through the logical expressions in order and executes the code corresponding to the first expression that is true and then stops. This means that the order of the expressions is vital to how `case_when` works. Note that the final statement in the example above is `TRUE ~ 0`. This plays the role equivalent to `else` in an `if ... else` statement. Because the expression to left of `~` obviously is always true, the code the right will be executed if and only if all other expressions are false.

The `case_when` function is more compact than nested *if ... else* statements. More usefully, it is vectorized. In other words, the `sawtooth` function, which uses nested *if ... else* statements, will not work properly if the input argument is a vector rather than a single value (which is, strictly speaking, a vector of length 1). By contrast, `sawtooth2`, which uses `case_when` will work will a vector input, as in the following example.

```
x <- c(0.1, 0.3, 0.5, 0.6, 0.7, 0.9)
sawtooth2(x)
#> [1] 0.3 0.9 0.5 0.8 0.1 0.7
```

Iterations

In R, there are two types of iterations or *loops*, which we informally refer to as *`for` loops* and *`while` loops*.

`for` loops

In order to understand for loops, let us re-use the `sawtooth` function above. We mentioned that this function can not be applied to vectors but only to single values. Let's say we had the following vector of 1000 elements to which we wished to apply the `sawtooth` function.

```
N <- 1000
x <- seq(-0.1, 1.1, length.out = N)
```

In principle, we could apply `sawtooth` to each element of `x`, one element at a time, as follows.

```
# Create a vector of 0's of same length as x
# This can also be done with `y <- vector('double', N)`
y <- numeric(N)

y[1] <- sawtooth(x[1])
y[2] <- sawtooth(x[2])
y[3] <- sawtooth(x[3])
...
y[N] <- sawtooth(x[N]) # where N = 1000
```

It should be obvious that we want to avoid this at all costs. Instead, we can create a for loop as follows.

```
for (i in 1:N) {
  y[i] <- sawtooth(x[i])
}
```

Essentially, this loop repeatedly executes the statement `y[i] <- sawtooth(x[i])`. On the first iteration, `i` takes the value of `1`. On the second iteration, `i` takes the value of `2`, and so on, until the final iteration where `i` takes the values of `N`. In other words, for each value of `i` from `1` to `N`, we execute `y[i] <- sawtooth(x[i])`. This is exactly equivalent to doing the following.

```
y[1] <- sawtooth(x[1])
y[2] <- sawtooth(x[2])
y[3] <- sawtooth(x[3])
...
y[N] <- sawtooth(x[N]) # where N = 1000
```

The general form of a for loop is as follows.

```
for (<var> in <sequence>) {
  <code body>
}
```

The for loop iteration begins with the `for` keyword followed by a round bracketed expression of the form `(<var> in <sequence>)` where `<var>` is what we'll call the *loop variable* and `<sequence>` is (usually) a

vector or list of items. After the round brackets is some code enclosed by `{}`. For each value in the `<sequence>`, the `<var>` is set to this value and the `<code body>` is executed. We could write this in pseudo-code as follows.

```
for each value in <sequence>
  set <var> equal to this value
  execute <code body>
```

In other words, the for loop executes the `<code body>` for each value in `<sequence>`, setting `<var>` to this value on each iteration.

Let's us look at some further examples.

Example 1: In the following, we take a list of people's names and print a greeting to them.

```
people <- c('bill', 'hillary', 'donald', 'george')
for (person in people){
  print(paste('Hello', person))
}
#> [1] "Hello bill"
#> [1] "Hello hillary"
#> [1] "Hello donald"
#> [1] "Hello george"
```

In this example, we execute `print(paste('Hello', person))` 4 times. On the first iteration, the value of `person` takes the value of `people[1]`, which is bill. On the second iteration, `person` takes the value of `people[2]`, which is hillary, and so on.

Example 2: The for loop in the previous example can also be implemented as follows.

```
for (i in seq_along(people)){
  print(paste('Hello', people[i]))
}
#> [1] "Hello bill"
#> [1] "Hello hillary"
#> [1] "Hello donald"
#> [1] "Hello george"
```

Here, `seq_along(people)` gives us the sequence of integers from 1 to `length(people)`. It is safer than doing, for example, `1:length(people)` because if `people` was in fact empty, `1:length(people)` would return 1, 0, while `seq_along(people)` would return an empty vector.

Example 3: Here, we sum all the elements in the vector `values`.

```
values <- c(51, 45, 53, 53, 46)
s <- 0
for (value in values){
  s <- s + value
}
s
#> [1] 248
```

Example 4: Here, we create a cumulative sum vector for `values`.

```
cumulative_values <- numeric(length(values))
for (i in seq_along(values)){
  if (i == 1){
    cumulative_values[i] <- values[i]
  }
  else {
    cumulative_values[i] <- cumulative_values[i - 1] + values[i]
```

```
  }
}
cumulative_values
#> [1]  51  96 149 202 248
```

Example 5: In this example, we implement the famous chaotic dynamical system described by May (1976). In this, the value of the system at time t is

$$x_t = r x_{t-1}(1 - x_{t-1}),$$

where, in its chaotic regime, r takes values approximately in the range $(3.5695, 4.0)$, and the initial value of the system is $x_1 \in (0, 1)$. In this example, we set $x_1 = 0.5$ and set $r = 3.75$, and iterate for 500 iterations.

```
N <- 500
r <- 3.75
x <- numeric(N)
x[1] <- 0.5
for (t in seq(2, N)){
  x[t] <- r * x[t-1] * (1 - x[t-1])
}
```

The plot of `x` for each value of `t` is shown in Figure 3.

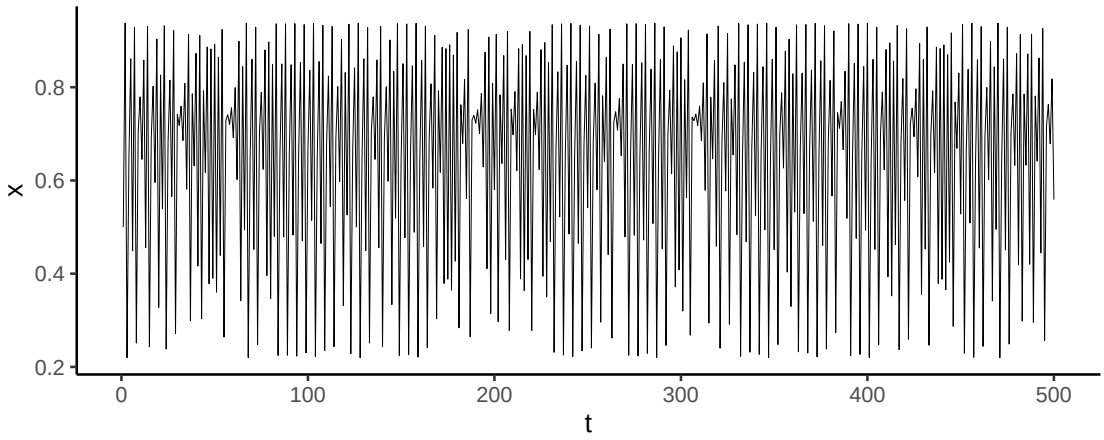


Figure 3: The time series of the chaotic logistic map. This time series can be obtained using a simple *for loop*.

Example 6: The following example illustrates how we can easily automate repetitive tasks, and save ourselves from considerable amount of time other spent on dull manual work. Let's say we have a large set of `.csv` files that we need to read in as data frames and then concatenate them into one large data frame.

```
file_names <- c('data_1.csv',
                'data_2.csv',
```

```
                'data_3.csv')
# create empty list
df_list <- vector('list', length(file_names))
# read in each file; store contents in list
for (i in seq_along(file_names)){
  df_list[[i]] <- read_csv(file_names[i])
}
# bind rows
data_df <- bind_rows(df_list)
```

In this example, there were just 3 files to read in and concatenate, but had there been hundreds or thousands of code, the required code would have been identical and the total running time might only be seconds or minutes.

`while` loops

Unlike for loops, which iterate through each value in a sequence of values, while loops continuously execute the code body while a condition remains true. The general form of a while loop is as follows.

```
while (<condition>) {
  <code body>
}
```

Here, `<condition>` is a logical expression that evaluates to `TRUE` or `FALSE`. The while loop will continue to execute `<code body>` as long as `<condition>` remains `TRUE`.

As a very simple example of a while loop, let's say we wish to find the largest value of k such that $2^k \leq 10^6$.

```
k <- 0
while (2^(k+1) <= 10^6) {
  k <- k + 1
}
k
#> [1] 19
```

Here, we see that we keep incrementing `k` by 1 until `2^(k+1)` is greater than 10^6. At that point, the condition `2^(k+1) <= 10^6` is false, and the *while loop* terminates. The final value of `k` is the highest of value of k such that $2^k \leq 10^6$.

While loops are very powerful. Anything that can be implemented by a for loop can also be implemented by a while loop, but the converse is not true. An an example of a while loop implementing a for loop, let's reimplement the summation for loop above.

```
values <- c(51, 45, 53, 53, 46)
i <- 1
s <- 0
while (i <= length(values)) {
  s <- s + values[i]
  i <- i + 1
}
s
#> [1] 248
```

A crucial feature of this while loop is the loop *counter* `i`, which is initialized to 1 and incremented by 1 on each iteration. This is a common feature of while loops.

Another common pattern in while loops is a conditional with a `break` statement. Whenever `break` occurs, the while loop terminates. Consider the following example where we again find the largest value of k such that $2^k \leq 10^6$.

```
k <- 0
while (TRUE){

  if (2^(k + 1) > 10^6){
    break
  }

  k <- k + 1
}

k
#> [1] 19
```

Notice that in this case the while loop's condition is always `TRUE`, and the conditional with the break statement is doing the crucial work of terminating the loop.

An equivalent of `while (TRUE)` is the `repeat` statement.

```
k <- 0
repeat {

  if (2^(k + 1) > 10^6){
    break
  }

  k <- k + 1
}
k
#> [1] 19
```

An example of a simple but nontrivial algorithm involving a while loop is where we sample from a discrete probability distribution. Consider the following probability distribution for a discrete random variable with 6 possible values.

```
p <- c(0.1, 0.2, 0.25, 0.15, 0.1, 0.2)
```

This tells us that the probability that the variable takes the value of 1 is 0.1, that it takes the value of 2 is 0.2, and so on. If we would like to sample from this probability distribution, we can do so using the following algorithm involving a while loop. First, we calculate the cumulative sum of `p`.

```
f <- cumsum(p)
f
#> [1] 0.10 0.30 0.55 0.70 0.80 1.00
```

Then, we sample uniformly at random from the interval $(0, 1)$, which we can do in R with `runif()`.

```
r <- runif(1)
```

Then we begin to step through each value of `f`, beginning with the first value, and test if `r` is less than or equal to this value. If it is, we stop. Otherwise, we move on to the next value in `f`.

```
k <- 1
while (TRUE) {

  if (r <= f[k]) {
    break
  }

  k <- k + 1
```

```
}
```

This sampler could also be implemented as follows.

```
k <- 1
while (r > f[k]) {
  k <- k + 1
}
```

Functionals

Functionals are functions that take a function as input and return a vector. They play an important role in programming in R, often taking the place of for loops. There are many functionals in the base R language. Here, we will look at the most useful or widely used ones.

`lapply`

One of the most widely used functionals in R is the `lapply` function, which takes two required arguments, a vector or list and a function, and then applies the function to each element in the vector or list and returns a new list. As an example, instead of using a for loop, we could use `lapply` to apply the `sawtooth` function used above to each element of a vector `x`. Here's the original for loop.

```
N <- 1000
x <- seq(-0.1, 1.1, length.out = N)
y <- numeric(N)
for (i in 1:N) {
  y[i] <- sawtooth(x[i])
}
```

We can replace the preassignment of `y`, i.e. `y <- numeric(N)`, and the for loop entirely using `lapply` as follows.

```
y <- lapply(x, sawtooth)
```

The `lapply` function returns a list with as many elements as there are elements in `x`.

```
length(x)
#> [1] 1000
length(y)
#> [1] 1000
class(y)
#> [1] "list"
head(y, 3) # first three elements of y
#> [[1]]
#> [1] 0
#>
#> [[2]]
#> [1] 0
#>
#> [[3]]
#> [1] 0
```

Should we prefer that `y` be a vector rather than a list, we can `unlist` it.

```
y <- unlist(y)
head(y, 3)
#> [1] 0 0 0
```

There are other options too for returning a vector from a functional, as we will see shortly.

As another application of `lapply`, consider the for loop example above where we read in data frames from multiple files using `read_csv` and stored them in a list that was then concatenated into `data_df`.

```
file_names <- c('data_1.csv',
                'data_2.csv',
                'data_3.csv')
df_list <- vector('list', length(file_names))
for (i in seq_along(file_names)){
  df_list[[i]] <- read_csv(file_names[i])
}
data_df <- bind_rows(df_list)
```

The preassignment to `df_list` and the for loop can be replaced with the use of `lapply`.

```
df_list <- lapply(file_names, read_csv)
```

The general form of the `lapply` function is as follows.

```
returned_list <- lapply(<vector or list>, <function>)
```

Sometimes, we may need to use a function that takes arguments in `lapply`. As an example, let us imagine that we wish to calculate the trimmed mean of each vector, and also ignoring missing values, in a list of vectors. The trimmed mean, which we described in Chapter 5, is where we compute the mean after a certain proportion of the high and low elements have been trimmed. The trimmed mean of a vector, where we remove 5% of values on the upper and lower extremes of the vector, and also ignoring missing values, is as follows.

```
x <- c(10, 20, NA, 125, 35, 15)
mean(x, trim = 0.05, na.rm = T)
#> [1] 41
```

To use the `trim = 0.05` and `na.rm = T` arguments when we use `lapply`, we supply them as optional arguments after the function name as follows.

```
data_vectors <- list(x = c(NA, 10, 11, 12, 1001, -20),
                     y = c(5, 10, 7, 2500, 6),
                     z = c(2, 4, 1000, 8, 5)
)
lapply(data_vectors, mean, trim = 0.20, na.rm = T) %>%
  unlist()
#>         x         y         z
#> 11.000000  7.666667  5.666667
```

Note that because data frames (and tibbles) are essentially lists of vectors, `lapply` can be easily used to apply a function to all columns of a data frame.

```
data_df <- tibble(x = rnorm(10),
                  y = rnorm(10),
                  z = rnorm(10))
lapply(data_df, mean) %>%
  unlist()
#>           x           y           z
#>  0.24504010 -0.43967492 -0.05399682
```

The above function accomplishes the same thing as the following `summarise_all` function, except that `summarize_all` returns a tibble.

```
summarise_all(data_df, mean)
#> # A tibble: 1 x 3
#>       x      y       z
```

```
#>   <dbl>  <dbl>   <dbl>
#> 1 0.245 -0.440 -0.0540
```

sapply and vapply

As we've seen, `lapply` always returns a list. Sometimes, this returned list is a list of single values or list of vectors of the same length and type. In these cases, it would be preferable to convert these lists to vectors or matrices. We saw this in the case of using `sawtooth` with `lapply` above, where we manually converted the returned list into a vector using `unlist`. Variants of `lapply`, `sapply` and `vapply`, can facilitate doing these conversions. The `sapply` function works like `lapply` but will attempt to simplify the list as a vector or a matrix if possible. In the following example, we use `sapply` to apply `sawtooth` to each element of `x`.

```
N <- 1000
x <- seq(-0.1, 1.1, length.out = N)
y <- sapply(x, sawtooth)
head(y, 5)
#> [1] 0 0 0 0 0
```

Here, because the list that would have been returned by `lapply`, had we used it here, is a list of length `N` of single numeric values (or numeric vectors of length 1), `sapply` can produce and return a numeric vector of length `N`.

If the results of `lapply` is a list of length `N` of numeric vectors of length 5, for example, then this list could be simplified to a matrix. In the following example, the list returned by `lapply` is a list like this.

```
data_df <- tibble(x = rnorm(100),
                  y = rnorm(100),
                  z = rnorm(100))
lapply(data_df, quantile)
#> $x
#>          0%         25%         50%         75%        100%
#> -2.31932737 -0.70960382 -0.05431694  0.72976278  2.13348636
#>
#> $y
#>          0%         25%         50%         75%        100%
#> -2.82300012 -0.48106048  0.04061442  0.61636766  2.13286973
#>
#> $z
#>         0%        25%        50%        75%       100%
#> -2.4662529 -0.6690109 -0.1111398  0.5306826  2.1873352
```

If we use `sapply` here instead, the result is a matrix.

```
sapply(data_df, quantile)
#>                 x           y          z
#> 0%   -2.31932737 -2.82300012 -2.4662529
#> 25%  -0.70960382 -0.48106048 -0.6690109
#> 50%  -0.05431694  0.04061442 -0.1111398
#> 75%   0.72976278  0.61636766  0.5306826
#> 100%  2.13348636  2.13286973  2.1873352
```

In cases where the list returned by `lapply` has elements of different lengths, `sapply` can not simplify it.

```
X <- list(x = rnorm(2),
          y = rnorm(3),
          z = rnorm(4)
)
sapply(X, function(x) x^2)
```

```
#> $x
#> [1] 0.0503656 0.1913169
#>
#> $y
#> [1] 1.7629691 0.2194715 1.7954867
#>
#> $z
#> [1] 2.7736662 0.2904671 0.9987699 0.1166150
```

The `vapply` function is a safer version of `sapply` because it specifies the nature of the returned values of each application of the function. For example, we know the default returned value of `quantile` will be a numeric vector of length 5. We can specify this as the `FUN.VALUE` argument to `vapply`.

```
vapply(data_df, quantile, FUN.VALUE=numeric(5))
#>                x           y          z
#> 0%   -2.31932737 -2.82300012 -2.4662529
#> 25%  -0.70960382 -0.48106048 -0.6690109
#> 50%  -0.05431694  0.04061442 -0.1111398
#> 75%   0.72976278  0.61636766  0.5306826
#> 100%  2.13348636  2.13286973  2.1873352
```

mapply and Map

The functionals `lapply`, `sapply`, `vapply` take a function and apply it to each element in a vector or list. In other words, each element in the vector or list is supplied as the argument to the function. While we may, as we described above, have other arguments to the function set to fixed values for each function application, we can not use `lapply`, `sapply` or `vapply` to apply functions to two or more lists or vectors at the same time. Consider the following function.

```
power <- function(x, k) x^k
```

If we had a vector of `x` values, and set `k` to e.g. 3, we could do the following.

```
x <- c(2, 3, 4, 5)
sapply(x, power, k=5)
#> [1]   32  243 1024 3125
```

However, if we had the vector of `x` values and a vector of `k` values, and which to apply each element of `x` and the corresponding element of `k` to `power`, we need to use `mapply`, as in the following example.

```
x <- c(2, 3, 4, 5)
k <- c(2, 3, 2, 2)
mapply(power, x = x, k = k)
#> [1]  4 27 16 25
```

As we can see, for each index `i` of `x` and `k`, we calculate `power(x[i], k[i])`. This `mapply` is therefore equivalent to the following for loop.

```
for (i in seq_along(x)){
  power(x[i], k[i])
}
```

With `mapply`, we can iterate over any number of lists of input arguments simultaneously. As an example, the random number generator `rnorm` function takes 3 arguments: `n`, `mean`, and `sd`. In the following, we apply `rnorm` to each value of of lists of these three arguments.

```
set.seed(101)
n <- c(2, 3, 5)
mu <- c(10, 100, 200)
sigma <- c(1, 10, 10)
```

```
mapply(rnorm, n = n, mean = mu, sd = sigma)
#> [[1]]
#> [1]  9.673964 10.552462
#>
#> [[2]]
#> [1]  93.25056 102.14359 103.10769
#>
#> [[3]]
#> [1] 211.7397 206.1879 198.8727 209.1703 197.7674
```

As we can see, we effectively execute `rnorm(n=3, mean=10, sd=1)`, `rnorm(n=5, mean=5, sd = 10)`, and so on.

The `Map` function works just like `mapply`, with minor differences, such as not ever simplifying the results.

```
set.seed(101)
Map(rnorm, n = n, mean = mu, sd = sigma)
#> [[1]]
#> [1]  9.673964 10.552462
#>
#> [[2]]
#> [1]  93.25056 102.14359 103.10769
#>
#> [[3]]
#> [1] 211.7397 206.1879 198.8727 209.1703 197.7674
```

As we can see, `Map` and `mapply` are identical in their usage and in what they do. However, by default, `mapply` will attempt to simply its output like `sapply`, if possible. We saw this with the use of the `power` function with `mapply` above. However, if we replace `mapply` with `Map`, no simplification is applied, and we obtain a list as output.

```
x <- c(2, 3, 4, 5)
k <- c(2, 3, 2, 2)
Map(power, x = x, k = k)
#> [[1]]
#> [1] 4
#>
#> [[2]]
#> [1] 27
#>
#> [[3]]
#> [1] 16
#>
#> [[4]]
#> [1] 25
```

`Filter`, `Find`, and `Position`

The `Filter` functional takes a *predicate*, which is a function that returns a logical value, and a vector or list and returns those elements of the list for which the predicate is true. As an example, here we have a data frame with three variables.

```
data_df <- tibble(x = rnorm(3),
                  y = rnorm(3),
                  z = c('a', 'b', 'c')
)
```

We can select the numeric vectors of `data_df` as follows.

```
Filter(is.numeric, data_df)
#> # A tibble: 3 x 2
#>        x      y
#>    <dbl>  <dbl>
#> 1  0.526 -1.47
#> 2 -0.795 -0.237
#> 3  1.43  -0.193
```

In this example, we are doing what could otherwise be accomplished with `dplyr`'s `select_if` function.

```
select_if(data_df, is.numeric)
#> # A tibble: 3 x 2
#>        x      y
#>    <dbl>  <dbl>
#> 1  0.526 -1.47
#> 2 -0.795 -0.237
#> 3  1.43  -0.193
```

Unlike `select_if`, `Filter` can be applied to data structure other than data frames. For example, in the following example, we select all elements of a random sample of 20 integers that are multiples of 3.

```
s <- sample.int(100, size=10, replace=T)
s
#>  [1] 31 79 51 14 67 42 50 43 14 25
Filter(function (x) x %% 3 == 0, s)
#> [1] 51 42
```

The `Find` function searches through a vector or list to find the first element for which a predicate is true.

```
Find(function (x) x %% 3 == 0, s)
#> [1] 51
```

If the predicate is not true of any element, then `NULL` is returned.

```
Find(function (x) x < 0, s)
#> NULL
```

The `Position` function is like `Find`, but returns the position of the first element for which the predicate is true.

```
Position(function (x) x %% 3 == 0, s)
#> [1] 3
```

Functionals with purrr

The `purrr` package in the `tidyverse` provides functionals like those just covered, but which have consistent with one another in terms of how they are used, and also with how other `tidyverse` functions are used. In addition, `purrr` provides additional functional tools beyond those in base R. We can load `purrr` with `library(purrr)`, but it is also loaded by `library(tidyverse)`.

map

One of the main tools in `purrr` is `map` and its variants. It is very similar to `lapply`. It takes a list (or vector) and a function, and applies the function to each element of the list. To re-use an example from above, in the following, we apply `read_csv` to each file name in a list of file names, and collect the data frames that are produced in a list `data_df_list`.

```
file_list <- c('data_1.csv',
               'data_2.csv',
```

```
              'data_3.csv')
data_df_list <- map(file_list, read_csv)
```

As with `lapply`, we can supply arguments for the function being applied as optional arguments. For example, if we wanted to read in the data files as in the previous example, but just read no more than 100 rows of data, supply the `n_max` argument to `read_csv` as follows.

```
data_df_list <- map(file_list, read_csv, n_max=100)
```

The `map` function have `_if` and `_at` variants, which function similarly to the `_if` and `_at` variants of the `dplyr` verbs we met earlier. For example, if we want to apply `read_csv` only if the name of the data file is not `data_2.csv`, we can use `map_if` as follows.

```
data_df_list <- map_if(file_list,
                       function(x) x != 'data_2.csv',
                       read_csv)
```

We could accomplish the same thing by using `map_at`, which can take positional arguments, or negative positions, to include or exclude elements of a list. For example, if we use `-2` as the second argument to `map_at` it will skip the second item in `file_list`.

```
data_df_list <- map_at(file_list,
                       -2,
                       read_csv)
```

When the list that is returned by `map` or its variants can be simplified to a vector, we can use the `map_dbl`, `map_int`, `map_lgl`, `map_chr` variants of `map` to simplify the list to a vector of doubles, integers, Booleans, or characters, respectively. In the following example, we apply functions that produce integers, logicals, doubles, or characters, to each column of a data frame.

```
data_df <- read_csv('data_1.csv')

map_int(data_df, length)
#>    x    y    z
#> 1000 1000 1000
map_lgl(data_df, is_integer)
#>     x     y     z
#> FALSE FALSE FALSE
map_dbl(data_df, mean)
#>             x             y             z
#> -0.0259532006  0.0643665838 -0.0009872201
map_chr(data_df, class)
#>         x         y         z
#> "numeric" "numeric" "numeric"
```

There is also a `map_df` that can be used when the list contains data frames that can be concatenated together. For example, to read in all three data frames and concatenate them we can do the following.

```
data_df_all <- map_df(file_list, read_csv)
```

What is accomplished with `map_df` here is exactly what would be obtained by a combination of `map` and `dplyr`'s `bind_rows`.

```
data_df_all_2 <- map(file_list, read_csv) %>%
  bind_rows()
all_equal(data_df_all, data_df_all_2)
#> [1] TRUE
```

The use of `map_df` in the above example is exactly equivalent to using `map_dfr`, which forces the creation of the data frame by row binding.

```
data_df_all_3 <- map_dfr(file_list, read_csv)
all_equal(data_df_all, data_df_all_3)
#> [1] TRUE
```

On the other hand, if we wanted to create a data frame by column binding, we could use `map_dfc`. It is possible to do this with the data frames in the previous examples because they all have equal numbers of rows.

```
data_df_all_4 <- map_dfc(file_list, read_csv)
```

The resulting data frame `data_df_all_4` is of the following dimensions.

```
dim(data_df_all_4)
#> [1] 1000    9
```

Its column names are as follows.

```
names(data_df_all_4)
#> [1] "x...1" "y...2" "z...3" "x...4" "y...5" "z...6" "x...7" "y...8" "z...9"
```

purrr style anonymous functions

We saw above that we can use anonymous functions in `lapply`, `sapply`, etc., functionals. This can be done in `purrr` functionals like the `map` family too, as we see in the following example.

```
map_dbl(data_df,
        function(x) mean(log(abs(x)))
)
#>          x          y          z
#> -0.6400092 -0.5868030 -0.5765645
```

However, `purrr` provides *syntactic sugar* to allow us to rewrite this as follows.

```
map_dbl(data_df,
        ~ mean(log(abs(.)))
)
#>          x          y          z
#> -0.6400092 -0.5868030 -0.5765645
```

In other words, in place of the `function(x)`, we have `~`, and in place of the anonymous function's input variable we have `.`.

map2 and pmap

When we have two or more than two sets of input arguments, we can use `map2` and `pmap`, respectively. Both of these functions also have the `_lgl`, `_int`, `_dbl`, `_chr`, `_df`, `_dfr`, `_dfc` variants that we saw with `map`.

As an example of a `map2` function, we'll use the `power` function that two input arguments.

```
x <- c(2, 3, 4, 5)
k <- c(2, 3, 2, 2)
map2_dbl(x, k, power)
#> [1]  4 27 16 25
```

As an example of a `pmap` function, we can reimplement the `rnorm` based sampler that we originally wrote with `mapply`. The first argument to `pmap` is a list whose length is the number of arguments being passed to the function. If want to iterate over different values of the `n`, `mean` and `sd` arguments for `rnorm`, as we did in the example above, we'd set up a list like the following.

```
args <- list(n = c(2, 3, 5),
             mean = c(10, 100, 200),
             sd = c(1, 10, 10))
```

We then use `pmap` as follows.

```
pmap(args, rnorm)
#> [[1]]
#> [1] 8.734939 9.313147
#>
#> [[2]]
#> [1]  95.54338 112.24082 103.59814
#>
#> [[3]]
#> [1] 204.0077 201.1068 194.4416 217.8691 204.9785
```

`walk`

The `walk` function in `purrr` is like `map` but is used with functions that are called just for their *side effects*. Put informally, a side effect of a function is an effect performed by a function other than its return value. A very common side effect of a function is writing something to a file. For example, while the `write_csv` function does return a value, it is simply the value of name of the file it is written. The real action done by `write_csv` is in its side effect. Like `map`, `walk` has has variants `walk2` and `pwalk` that are exactly analogous to `map2` and `pmap`.

As an example, let us perform the inverse of `map(file_list, read_csv)` in the following code, which takes a list of `.csv` files, reads in their data as data frames, and then stores them in the list `data_df_list`.

```
file_list <- c('data_1.csv',
               'data_2.csv',
               'data_3.csv')
data_df_list <- map(file_list, read_csv)
```

The inverse should take each element in the list and write it back to the appropriately named file. For these, we can use `walk2`, which will iterate over the data frames in the list and the names in `file_list`.

```
walk2(data_df_list, file_list, write_csv)
```

`keep` and `discard`

The `purrr` package also provides us with workalike to base R's `Filter`. For example, if we want to select out elements of a list or vector for which some predicate is true, we can use `keep`. To select those elements for which the predicate is not true, we can use `discard`. As an example, here we select out the even numbers in a seq of integers from 1 to 20.

```
keep(seq(20), ~ . %% 2 == 0)
#>  [1]  2  4  6  8 10 12 14 16 18 20
```

To select the odds numbers, which are obviously not the even numbers, we can use `discard` as follows.

```
discard(seq(20), ~ . %% 2 == 0)
#>  [1]  1  3  5  7  9 11 13 15 17 19
```

References

May, Robert M. 1976. "Simple Mathematical Models with Very Complicated Dynamics." *Nature* 261 (5560): 459–67.

Wickham, Hadley. 2019. *Advanced R*. 2nd ed. Chapman; Hall/CRC.