Attributes

All objects can have arbitrary additional attributes, used to store metadata about the object. Attributes can be thought of as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

```
y <- 1:10
attr(y, "my_attribute") <- "This is a vector"
attr(y, "my_attribute")
#> [1] "This is a vector"
str(attributes(y))
#> List of 1
#>  $ my_attribute: chr "This is a vector"
```

The `structure()` function returns a new object with modified attributes:

```
structure(1:10, my_attribute = "This is a vector")
#>  [1]  1  2  3  4  5  6  7  8  9 10
#> attr(,"my_attribute")
#> [1] "This is a vector"
```

By default, most attributes are lost when modifying a vector:

```
attributes(y[1])
#> NULL
attributes(sum(y))
#> NULL
```

The only attributes not lost are the three most important:

- Names, a character vector giving each element a name, described in names.
- Dimensions, used to turn vectors into matrices and arrays, described inmatrices and arrays.
- Class, used to implement the S3 object system, described in S3.

Each of these attributes has a specific accessor function to get and set values. When working with these attributes, use `names(x)`, `dim(x)`, and `class(x)`, not `attr(x, "names")`, `attr(x, "dim")`, and `attr(x, "class")`.

Names

You can name a vector in three ways:

- When creating it: `x <- c(a = 1, b = 2, c = 3)`.
- By modifying an existing vector in place: `x <- 1:3; names(x) <- c("a", "b", "c")`.
- By creating a modified copy of a vector: `x <- setNames(1:3, c("a", "b", "c"))`.

Names don't have to be unique. However, character subsetting, described insubsetting, is the most important reason to use names and it is most useful when the names are unique.

Not all elements of a vector need to have a name. If some names are missing, `names()` will return an empty string for those elements. If all names are missing, `names()` will return `NULL`.

```
y <- c(a = 1, 2, 3)
names(y)
#> [1] "a" ""  ""

z <- c(1, 2, 3)
names(z)
#> NULL
```

You can create a new vector without names using `unname(x)`, or remove names in place with `names(x) <- NULL`.

Factors

One important use of attributes is to define factors. A factor is a vector that can contain only predefined values, and is used to store categorical data. Factors are built on top of integer vectors using two attributes: the `class()`, "factor", which makes them behave differently from regular integer vectors, and the `levels()`, which defines the set of allowed values.

```
x <- factor(c("a", "b", "b", "a"))
x
#> [1] a b b a
#> Levels: a b
class(x)
#> [1] "factor"
levels(x)
#> [1] "a" "b"

# You can't use values that are not in the levels
x[2] <- "c"
#> Warning in `[<-.factor`(`*tmp*`, 2, value = "c"): invalid factor level, NA
#> generated
x
#> [1] a    <NA> b    a
#> Levels: a b

# NB: you can't combine factors
c(factor("a"), factor("b"))
#> [1] 1 1
```

Factors are useful when you know the possible values a variable may take, even if you don't see all values in a given

dataset. Using a factor instead of a character vector makes it obvious when some groups contain no observations:

```
sex_char <- c("m", "m", "m")
sex_factor <- factor(sex_char, levels = c("m", "f"))

table(sex_char)
#> sex_char
#> m
#> 3
table(sex_factor)
#> sex_factor
#> m f
#> 3 0
```

Sometimes when a data frame is read directly from a file, a column you'd thought would produce a numeric vector instead produces a factor. This is caused by a non-numeric value in the column, often a missing value encoded in a special way like `.` or `-`. To remedy the situation, coerce the vector from a factor to a character vector, and then from a character to a double vector. (Be sure to check for missing values after this process.) Of course, a much better plan is to discover what caused the problem in the first place and fix that; using the `na.strings` argument to `read.csv()` is often a good place to start.

```
# Reading in "text" instead of from a file here:
z <- read.csv(text = "value\n12\n1\n.\n9")
typeof(z$value)
#> [1] "integer"
as.double(z$value)
#> [1] 3 2 1 4
# Oops, that's not right: 3 2 1 4 are the levels of a factor,
# not the values we read in!
class(z$value)
#> [1] "factor"
# We can fix it now:
as.double(as.character(z$value))
#> Warning: NAs introduced by coercion
#> [1] 12  1 NA  9
# Or change how we read it in:
z <- read.csv(text = "value\n12\n1\n.\n9", na.strings=".")
typeof(z$value)
#> [1] "integer"
class(z$value)
#> [1] "integer"
z$value
#> [1] 12  1 NA  9
# Perfect! :)
```

Unfortunately, most data loading functions in R automatically convert character vectors to factors. This is suboptimal, because there's no way for those functions to know the set of all possible levels or their optimal order. Instead, use the argument `stringsAsFactors = FALSE` to suppress this behaviour, and then manually convert character vectors to factors using your knowledge of the data. A global option, `options(stringsAsFactors = FALSE)`, is available to control this behaviour, but I don't recommend using it. Changing a global option may have unexpected consequences when combined with other code (either from packages, or code that you're `source()`ing), and global options make code harder to understand because they increase the number of lines you need to read to understand how a single line of code will behave.

While factors look (and often behave) like character vectors, they are actually integers. Be careful when treating them like strings. Some string methods (like `gsub()` and `grepl()`) will coerce factors to strings, while others (like `nchar()`) will throw an error, and still others (like `c()`) will use the underlying integer values. For this reason, it's usually best to explicitly convert factors to character vectors if you need string-like behaviour. In early versions of R, there was a memory advantage to using factors instead of character vectors, but this is no longer the case.

Matrices and arrays

Adding a `dim()` attribute to an atomic vector allows it to behave like a multi-dimensional **array**. A special case of the array is the **matrix**, which has two dimensions. Matrices are used commonly as part of the mathematical machinery of statistics. Arrays are much rarer, but worth being aware of.

Matrices and arrays are created with `matrix()` and `array()`, or by using the assignment form of `dim()`:

```
# Two scalar arguments to specify rows and columns
a <- matrix(1:6, ncol = 3, nrow = 2)
# One vector argument to describe all dimensions
b <- array(1:12, c(2, 3, 2))

# You can also modify an object in place by setting dim()
c <- 1:6
dim(c) <- c(3, 2)
c
#>      [,1] [,2]
#> [1,]    1    4
#> [2,]    2    5
#> [3,]    3    6
dim(c) <- c(2, 3)
c
#>      [,1] [,2] [,3]
#> [1,]    1    3    5
#> [2,]    2    4    6
```

`length()` and `names()` have high-dimensional generalisations:

- `length()` generalises to `nrow()` and `ncol()` for matrices, and `dim()` for arrays.
- `names()` generalises to `rownames()` and `colnames()` for matrices, and `dimnames()`, a list of character vectors, for arrays.

```
length(a)
#> [1] 6
nrow(a)
#> [1] 2
ncol(a)
#> [1] 3
rownames(a) <- c("A", "B")
colnames(a) <- c("a", "b", "c")
a
#>   a b c
#> A 1 3 5
```

```
#> B 2 4 6

length(b)
#> [1] 12
dim(b)
#> [1] 2 3 2
dimnames(b) <- list(c("one", "two"), c("a", "b", "c"), c("A", "B"))
b
#> , , A
#>
#>     a b c
#> one 1 3 5
#> two 2 4 6
#>
#> , , B
#>
#>     a  b  c
#> one 7  9 11
#> two 8 10 12
```

`c()` generalises to `cbind()` and `rbind()` for matrices, and to `abind()` (provided by the `abind` package) for arrays. You can transpose a matrix with `t()`; the generalised equivalent for arrays is `aperm()`.

You can test if an object is a matrix or array using `is.matrix()` and `is.array()`, or by looking at the length of the `dim()`. `as.matrix()` and `as.array()` make it easy to turn an existing vector into a matrix or array.

Vectors are not the only 1-dimensional data structure. You can have matrices with a single row or single column, or arrays with a single dimension. They may print similarly, but will behave differently. The differences aren't too important, but it's useful to know they exist in case you get strange output from a function (`tapply()` is a frequent offender). As always, use `str()` to reveal the differences.

```
str(1:3)                   # 1d vector
#>  int [1:3] 1 2 3
str(matrix(1:3, ncol = 1)) # column vector
#>  int [1:3, 1] 1 2 3
str(matrix(1:3, nrow = 1)) # row vector
#>  int [1, 1:3] 1 2 3
str(array(1:3, 3))         # "array" vector
#>  int [1:3(1d)] 1 2 3
```

While atomic vectors are most commonly turned into matrices, the dimension attribute can also be set on lists to make list-matrices or list-arrays:

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <- c(2, 2)
l
```

```
#>      [,1]      [,2]
#> [1,] Integer,3 TRUE
#> [2,] "a"       1
```

These are relatively esoteric data structures, but can be useful if you want to arrange objects into a grid-like structure. For example, if you're running models on a spatio-temporal grid, it might be natural to preserve the grid structure by storing the models in a 3d array.

Functional programming

R, at its heart, is a functional programming (FP) language. This means that it provides many tools for the creation and manipulation of functions. In particular, R has what's known as first class functions. You can do anything with functions that you can do with vectors: you can assign them to variables, store them in lists, pass them as arguments to other functions, create them inside functions, and even return them as the result of a function.

The chapter starts by showing a motivating example, removing redundancy and duplication in code used to clean and summarise data. Then you'll learn about the three building blocks of functional programming: anonymous functions, closures (functions written by functions), and lists of functions. These pieces are twined together in the conclusion which shows how to build a suite of tools for numerical integration, starting from very simple primitives. This is a recurring theme in FP: start with small, easy-to-understand building blocks, combine them into more complex structures, and apply them with confidence.

The discussion of functional programming continues in the following two chapters:functionals explores functions that take functions as arguments and return vectors as output, and function operators explores functions that take functions as input and return them as output.

Outline

- Motivation motivates functional programming using a common problem: cleaning and summarising data before serious analysis.
- Anonymous functions shows you a side of functions that you might not have known about: you can use functions without giving them a name.
- Closures introduces the closure, a function written by another function. A closure can access its own arguments, and variables defined in its parent.
- Lists of functions shows how to put functions in a list, and explains why you might care.
- Numerical integration concludes the chapter with a case study that uses anonymous functions, closures and lists of functions to build a flexible toolkit for numerical integration.

Prequisites

You should be familiar with the basic rules of lexical scoping, as described in lexical scoping. Make sure you've installed the pryr package with `install.packages("pryr")`

Motivation

Imagine you've loaded a data file, like the one below, that uses −99 to represent missing values. You want to replace all the −99s with `NA`s.

```
# Generate a sample dataset
set.seed(1014)
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df
```

```
#>    a  b c   d   e f
#> 1  1  6 1   5 -99 1
#> 2 10  4 4 -99   9 3
#> 3  7  9 5   4   1 4
#> 4  2  9 3   8   6 8
#> 5  1 10 5   9   8 6
#> 6  6  2 1   3   8 5
```

When you first started writing R code, you might have solved the problem with copy-and-paste:

```
df$a[df$a == -99] <- NA
df$b[df$b == -99] <- NA
df$c[df$c == -98] <- NA
df$d[df$d == -99] <- NA
df$e[df$e == -99] <- NA
df$f[df$g == -99] <- NA
```

One problem with copy-and-paste is that it's easy to make mistakes. Can you spot the two in the block above? These mistakes are inconsistencies that arose because we didn't have an authorative description of the desired action (replace -99 with `NA`). Duplicating an action makes bugs more likely and makes it harder to change code. For example, if the code for a missing value changes from -99 to 9999, you'd need to make the change in multiple places.

To prevent bugs and to make more flexible code, adopt the "do not repeat yourself", or DRY, principle. Popularised by the "pragmatic programmers", Dave Thomas and Andy Hunt, this principle states: "every piece of knowledge must have a single, unambiguous, authoritative representation within a system". FP tools are valuable because they provide tools to reduce duplication.

We can start applying FP ideas by writing a function that fixes the missing values in a single vector:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
df$f <- fix_missing(df$e)
```

This reduces the scope of possible mistakes, but it doesn't eliminate them: you can no longer accidentally type -98 instead of -99, but you can still mess up the name of variable. The next step is to remove this possible source of error by combining two functions. One function, `fix_missing()`, knows how to fix a single vector; the other, `lapply()`, knows how to do something to each column in a data frame.

`lapply()` takes three inputs: `x`, a list; `f`, a function; and `...`, other arguments to pass to `f()`. It applies the function to

each element of the list and returns a new list. `lapply(x, f, ...)` is equivalent to the following for loop:

```
out <- vector("list", length(x))
for (i in seq_along(x)) {
  out[[i]] <- f(x[[i]], ...)
}
```

The real `lapply()` is rather more complicated since it's implemented in C for efficiency, but the essence of the algorithm is the same. `lapply()` is called a **functional**, because it takes a function as an argument. Functionals are an important part of functional programming. You'll learn more about them in functionals.

We can apply `lapply()` to this problem because data frames are lists. We just need a neat little trick to make sure we get back a data frame, not a list. Instead of assigning the results of `lapply()` to `df`, we'll assign them to `df[]`. R's usual rules ensure that we get a data frame, not a list. (If this comes as a surprise, you might want to read subsetting and assignment.) Putting these pieces together gives us:

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df[] <- lapply(df, fix_missing)
```

This code has five advantages over copy and paste:

- It's more compact.
- If the code for a missing value changes, it only needs to be updated in one place.
- It works for any number of columns. There is no way to accidentally miss a column.
- There is no way to accidentally treat one column differently than another.
- It is easy to generalise this technique to a subset of columns:

  ```
  df[1:5] <- lapply(df[1:5], fix_missing)
  ```

The key idea is function composition. Take two simple functions, one which does something to every column and one which fixes missing values, and combine them to fix missing values in every column. Writing simple functions that can be understood in isolation and then composed is a powerful technique.

What if different columns used different codes for missing values? You might be tempted to copy-and-paste:

```
fix_missing_99 <- function(x) {
  x[x == -99] <- NA
  x
}
fix_missing_999 <- function(x) {
  x[x == -999] <- NA
```

```
  x
}
fix_missing_9999 <- function(x) {
  x[x == -999] <- NA
  x
}
```

As before, it's easy to create bugs. Instead we could use closures, functions that make and return functions. Closures allow us to make functions based on a template:

```
missing_fixer <- function(na_value) {
  function(x) {
    x[x == na_value] <- NA
    x
  }
}
fix_missing_99 <- missing_fixer(-99)
fix_missing_999 <- missing_fixer(-999)

fix_missing_99(c(-99, -999))
#> [1]   NA -999
fix_missing_999(c(-99, -999))
#> [1] -99  NA
```

Extra argument

In this case, you could argue that we should just add another argument:

```
fix_missing <- function(x, na.value) {
  x[x == na.value] <- NA
  x
}
```

That's a reasonable solution here, but it doesn't always work well in every situation. We'll see more compelling uses for closures in MLE.

Now consider a related problem. Once you've cleaned up your data, you might want to compute the same set of numerical summaries for each variable. You could write code like this:

```
mean(df$a)
median(df$a)
sd(df$a)
mad(df$a)
IQR(df$a)
```

```
mean(df$b)
median(df$b)
sd(df$b)
mad(df$b)
IQR(df$b)
```

But again, you'd be better off identifying and removing duplicate items. Take a minute or two to think about how you might tackle this problem before reading on.

One approach would be to write a summary function and then apply it to each column:

```
summary <- function(x) {
  c(mean(x), median(x), sd(x), mad(x), IQR(x))
}
lapply(df, summary)
```

That's a great start, but there's still some duplication. It's easier to see if we make the summary function more realistic:

```
summary <- function(x) {
 c(mean(x, na.rm = TRUE),
   median(x, na.rm = TRUE),
   sd(x, na.rm = TRUE),
   mad(x, na.rm = TRUE),
   IQR(x, na.rm = TRUE))
}
```

All five functions are called with the same arguments (`x` and `na.rm`) repeated five times. As always, duplication makes our code fragile: it's easier to introduce bugs and harder to adapt to changing requirements.

To remove this source of duplication, you can take advantage of another functional programming technique: storing functions in lists.

```
summary <- function(x) {
  funs <- c(mean, median, sd, mad, IQR)
  lapply(funs, function(f) f(x, na.rm = TRUE))
}
```

This chapter discusses these techniques in more detail. But before you can start learning them, you need to learn the simplest FP tool, the anonymous function.

Anonymous functions

In R, functions are objects in their own right. They aren't automatically bound to a name. Unlike many languages (e.g., C, C++, Python, and Ruby), R doesn't have a special syntax for creating a named function: when you create a function, you use the regular assignment operator to give it a name. If you choose not to give the function a name, you get

an **anonymous function**.

You use an anonymous function when it's not worth the effort to give it a name:

```
lapply(mtcars, function(x) length(unique(x)))
Filter(function(x) !is.numeric(x), mtcars)
integrate(function(x) sin(x) ^ 2, 0, pi)
```

Like all functions in R, anonymous functions have `formals()`, a `body()`, and a parent `environment()`:

```
formals(function(x = 4) g(x) + h(x))
#> $x
#> [1] 4
body(function(x = 4) g(x) + h(x))
#> g(x) + h(x)
environment(function(x = 4) g(x) + h(x))
#> <environment: R_GlobalEnv>
```

You can call an anonymous function without giving it a name, but the code is a little tricky to read because you must use parentheses in two different ways: first, to call a function, and second to make it clear that you want to call the anonymous function itself, as opposed to calling a (possibly invalid) function *inside* the anonymous function:

```
# This does not call the anonymous function.
# (Note that "3" is not a valid function.)
function(x) 3()
#> function(x) 3()

# With appropriate parenthesis, the function is called:
(function(x) 3)()
#> [1] 3

# So this anonymous function syntax
(function(x) x + 3)(10)
#> [1] 13

# behaves exactly the same as
f <- function(x) x + 3
f(10)
#> [1] 13
```

You can call anonymous functions with named arguments, but doing so is a good sign that your function needs a name.

One of the most common uses for anonymous functions is to create closures, functions made by other functions. Closures are described in the next section.

Closures

> “An object is data with functions. A closure is a function with data.” — John D. Cook

One use of anonymous functions is to create small functions that are not worth naming. Another important use is to create closures, functions written by functions. Closures get their name because they **enclose** the environment of the parent function and can access all its variables. This is useful because it allows us to have two levels of parameters: a parent level that controls operation and a child level that does the work.

The following example uses this idea to generate a family of power functions in which a parent function (`power()`) creates two child functions (`square()` and `cube()`).

```
power <- function(exponent) {
  function(x) {
    x ^ exponent
  }
}

square <- power(2)
square(2)
#> [1] 4
square(4)
#> [1] 16

cube <- power(3)
cube(2)
#> [1] 8
cube(4)
#> [1] 64
```

When you print a closure, you don’t see anything terribly useful:

```
square
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x3a50680>
cube
#> function(x) {
#>     x ^ exponent
#>   }
#> <environment: 0x37f10f8>
```

That's because the function itself doesn't change. The difference is the enclosing environment, `environment(square)`. One way to see the contents of the environment is to convert it to a list:

```
as.list(environment(square))
#> $exponent
#> [1] 2
as.list(environment(cube))
#> $exponent
#> [1] 3
```

Another way to see what's going on is to use `pryr::unenclose()`. This function replaces variables defined in the enclosing environment with their values:

```
library(pryr)
unenclose(square)
#> function (x)
#> {
#>     x^2
#> }
unenclose(cube)
#> function (x)
#> {
#>     x^3
#> }
```

The parent environment of a closure is the execution environment of the function that created it, as shown by this code:

```
power <- function(exponent) {
  print(environment())
  function(x) x ^ exponent
}
zero <- power(0)
#> <environment: 0x4197fa8>
environment(zero)
#> <environment: 0x4197fa8>
```

The execution environment normally disappears after the function returns a value. However, functions capture their enclosing environments. This means when function a returns function b, function b captures and stores the execution environment of function a, and it doesn't disappear. (This has important consequences for memory use, see memory usage for details.)

In R, almost every function is a closure. All functions remember the environment in which they were created, typically either the global environment, if it's a function that you've written, or a package environment, if it's a function that someone else has written. The only exception is primitive functions, which call C code directly and don't have an associated environment.

Closures are useful for making function factories, and are one way to manage mutable state in R.

Function factories

A function factory is a factory for making new functions. We've already seen two examples of function factories, `missing_fixer()` and `power()`. You call it with arguments that describe the desired actions, and it returns a function that will do the work for you. For `missing_fixer()` and `power()`, there's not much benefit in using a function factory instead of a single function with multiple arguments. Function factories are most useful when:

- The different levels are more complex, with multiple arguments and complicated bodies.
- Some work only needs to be done once, when the function is generated.

Function factories are particularly well suited to maximum likelihood problems, and you'll see a more compelling use of them in mathematical functionals.

Mutable state

Having variables at two levels allows you to maintain state across function invocations. This is possible because while the execution environment is refreshed every time, the enclosing environment is constant. The key to managing variables at different levels is the double arrow assignment operator (`<<-`). Unlike the usual single arrow assignment (`<-`) that always assigns in the current environment, the double arrow operator will keep looking up the chain of parent environments until it finds a matching name. (Binding names to values has more details on how it works.)

Together, a static parent environment and `<<-` make it possible to maintain state across function calls. The following example shows a counter that records how many times a function has been called. Each time `new_counter` is run, it creates an environment, initialises the counter `i` in this environment, and then creates a new function.

```
new_counter <- function() {
  i <- 0
  function() {
    i <<- i + 1
    i
  }
}
```

The new function is a closure, and its enclosing environment is the environment created when `new_counter()` is run. Ordinarily, function execution environments are temporary, but a closure maintains access to the environment in which it was created. In the example below, closures `counter_one()` and `counter_two()` each get their own enclosing environments when run, so they can maintain different counts.

```
counter_one <- new_counter()
counter_two <- new_counter()

counter_one()
#> [1] 1
counter_one()
```

```
#> [1] 2
counter_two()
#> [1] 1
```

The counters get around the “fresh start” limitation by not modifying variables in their local environment. Since the changes are made in the unchanging parent (or enclosing) environment, they are preserved across function calls.

What happens if you don’t use a closure? What happens if you use `<-` instead of`<<-`? Make predictions about what will happen if you replace `new_counter()` with the variants below, then run the code and check your predictions.

```
i <- 0
new_counter2 <- function() {
  i <<- i + 1
  i
}
new_counter3 <- function() {
  i <- 0
  function() {
    i <- i + 1
    i
  }
}
```

Modifying values in a parent environment is an important technique because it is one way to generate “mutable state” in R. Mutable state is normally hard because every time it looks like you’re modifying an object, you’re actually creating and then modifying a copy. However, if you do need mutable objects and your code is not very simple, it’s usually better to use reference classes, as described in RC.

The power of closures is tightly coupled with the more advanced ideas in functionalsand function operators. You’ll see many more closures in those two chapters. The following section discusses the third technique of functional programming in R: the ability to store functions in a list.