Modeling Data With Functional Programming In R
Contents

1 Functions as a lingua franca 1
   1.1 First-Class Functions ........................................ 2
   1.2 Higher-order functions ...................................... 4
       1.2.1 Functions that take functions as arguments .......... 6
       1.2.2 Functions that return functions ..................... 9
   1.3 Closures .................................................... 10
       1.3.1 Interface compatibility .................................. 12
       1.3.2 State representation .................................... 14
       1.3.3 Mutable state ............................................. 17
       1.3.4 Generators ............................................... 19
   1.4 Functions In Mathematics ................................. 20
   1.5 A lambda calculus primer ................................ 22
       1.5.1 Reducible expressions ................................. 23
   1.6 Church numerals ........................................... 25
   1.7 Summary .................................................... 28

2 Vector Mechanics ............................................ 29
   2.1 Vectors as a polymorphic data type ..................... 31
       2.1.1 Vector construction .................................... 33
       2.1.2 Scalars .................................................. 34
       2.1.3 Atomic types ............................................ 35
       2.1.4 Coercion ............................................... 36
       2.1.5 Concatenation ............................................ 37
   2.2 Set theory ................................................... 42
       2.2.1 The empty set ......................................... 43
       2.2.2 Set membership ........................................ 44
       2.2.3 Set comprehensions and logic operations ............ 45
       2.2.4 Set complements ........................................ 46
   2.3 Indexing and subsequences ............................... 48
       2.3.1 Named indices ......................................... 49
       2.3.2 Logical indexing ....................................... 51
       2.3.3 Ordinal mappings ..................................... 52
       2.3.4 Sorting ................................................ 53
   2.4 Recycling .................................................. 54
   2.5 Summary .................................................... 56
3 Map Vectorization
  3.1 Map operations ............................................. 57
      3.1.1 Typed operations and the apply family ................. 62
  3.2 Preservation of Cardinality .................................. 62
      3.2.1 Sorting .............................................. 62
      3.2.2 NULL values ........................................ 62
  3.3 Transitive aspects of vectorization .......................... 62
  3.4 Semantics of map ........................................... 63
      3.4.1 Map vectorization .................................... 63
      3.4.2 Map equivalent form .................................. 63

4 Fold Vectorization ............................................... 65
  4.1 Fold operations .............................................. 66
      4.1.1 Commutativity of fold ................................ 70
  4.2 Semantics of fold ............................................ 70
      4.2.1 Fold vectorization .................................... 70
      4.2.2 Fold equivalent form .................................. 70
  4.3 Sequences, series, and closures .............................. 70
      4.3.1 Error analysis and the Euler-MacLaurin formula .......... 74
      4.3.2 Example: Taylor series approximation ................... 74

5 Canonical Data Structures ........................................ 79
  5.1 Matrices ................................................... 79
      5.1.1 Subsets and indexing .................................. 81
      5.1.2 Example: Numerical integration ........................ 85
  5.2 Lists ...................................................... 87
      5.2.1 Lists as untyped sets ................................ 88
      5.2.2 Lists as function arguments ............................ 88
  5.3 Data.frames ................................................ 88
      5.3.1 Concatenation in 2 dimensions .......................... 89

6 Higher-Order Functions in 2-Dimensions .......................... 91
  6.1 Map in 2 dimensions ........................................ 91
  6.2 Fold in 2 dimensions ....................................... 91

7 Transformations and Isomorphisms .................................. 93
  7.1 Windows and ranges ........................................ 93
  7.2 Blocks .................................................... 93
  7.3 Matrices to arrays ......................................... 93

8 Data Normalization ............................................... 95
  8.1 Normalization ............................................... 98
      8.1.1 Isolating data management logic ......................... 100
      8.1.2 Data sources and I/O abstraction ....................... 101
      8.1.3 Data formats and versions .............................. 103
      8.1.4 Example: Portfolio construction from a set of signals ... 104
8.1.5 Bad data .................................................. 104
8.1.6 Example: Corporate actions .......................... 104
8.2 Model artifacts .............................................. 105
  8.2.1 Model adjustments and new data requirements .... 105
  8.2.2 Back testing and validation artifacts ............... 105
  8.2.3 Example: Signal generation and portfolio optimization 106
  8.2.4 Visualization and reporting artifacts ............... 106
  8.2.5 Example: Saving charts with png .................. 107
8.3 Data artifacts ............................................. 107
  8.3.1 Iterative development and data formats .......... 108
  8.3.2 Preventing data dependencies ....................... 108
  8.3.3 Duck typing as a workaround ....................... 108
  8.3.4 Guards as a temporary code stash ................. 109
8.4 Persistence ............................................... 109
  8.4.1 Databases ........................................... 110
  8.4.2 Files .............................................. 110

9 Types ......................................................... 113
  9.1 Contextualization via types ............................ 115
    9.1.1 Bounded sets ...................................... 115
    9.1.2 Encapsulation of model methodology ............. 115
    9.1.3 Finite state machines and types ................. 115
  9.2 Revisiting set theory .................................. 115
    9.2.1 Constructors ...................................... 115
    9.2.2 Types and data structures ....................... 115
    9.2.3 The $ operator .................................... 116
    9.2.4 The @ operator .................................... 116
    9.2.5 Hierarchies and compatibility ................. 116
  9.3 Type constraints ....................................... 116
    9.3.1 Referential transparency and typed functions ... 116
    9.3.2 Type variables .................................... 116
    9.3.3 The range operators and types ................. 116
  9.4 Example: Binomial asset pricing model .............. 116

10 Model Specification And Modularity .......................... 117
  10.1 Polymorphic models and type hierarchies ............ 118
    10.1.1 Switching models ................................. 118
    10.1.2 Parametric models ................................ 118
    10.1.3 Stochastic models ................................ 118
    10.1.4 Example: Covariance matrix estimation .......... 118
    10.1.5 Example: Optimization and the simplex method 118
  10.2 Closures as model glue ................................ 119
    10.2.1 API consistency ................................... 119
    10.2.2 Avoiding interface leakage ...................... 119
    10.2.3 Example: Unconstrained optimization ........... 119
### List of Figures

1.1 Generalization of iris statistics using object-oriented programming ........................................ 5
1.2 Using the Strategy design pattern .................................................................................. 6
1.3 Iris classification with added features ................................................................. 7
1.4 Custom handling of NAs .......................................................................................... 7
1.5 A generalized classification function for iris data ............................................. 9
1.6 A higher-order function used to configure a SVM ......................................... 10
1.7 A class hierarchy for classification models .................................................... 14
1.8 A resource management function .................................................................. 15
1.9 Using a closure to manage external resources .............................................. 16
1.10 Evaluating the numerical stability of SVMs .................................................. 17
1.11 Classification error for SVM .............................................................. 18
1.12 A simple generator function ........................................................................ 19
1.13 Mapping Church numerals to natural numbers ........................................... 26
1.14 Addition for Church numerals ..................................................................... 27

2.1 Partial type coercion hierarchy for concatenation ........................................ 37
2.2 Loading the diabetes dataset .............................................................................. 41
2.3 Representing data as sets .................................................................................. 43
2.4 Convert a time into a decimal hour ............................................................... 55

4.1 Definition of the factorial function ................................................................ 70
4.2 Approximation of \( \cos(x) \) about \( x = 2 \) ...................................................... 76

5.1 Map for 2 dimensions ...................................................................................... 82
5.2 Fold for 2 dimensions ...................................................................................... 84

9.1 Using apply as a map between two sets ....................................................... 114

12.1 Using Monte Carlo to 'prove' the induction step ......................................... 125
12.2 Distribution of trials in Monte Carlo simulation ........................................ 125
12.3 Using Monte Carlo to 'prove' the induction step ......................................... 127
12.4 Distribution of trials in Monte Carlo simulation ......................................... 127
List of Tables

2.1 Logical operators given input(s) of length $n$ .................. 46
4.1 Common functions and their archetypes .................... 66
As a language paradigm, functional programming is not language-specific. Rather, functional programming is a theory for structuring programs based on function composition. In addition to function composition, functional programming is comprised of a (mostly) standard set of syntactic and semantic features. Many of these concepts originate from the lambda calculus, a mathematical framework for describing computation via functions. While each functional language supports a slightly different set of features, there is a minimal set of overlapping concepts that we can consider to form the basis of functional programming. This set consists of first-class functions, higher-order functions, and closures. Once these concepts are mastered, it is easy to identify and apply them in any language. In principle this is the same as learning the syntax of a new language: you begin by looking for the delimiter for statements, expressions, and blocks as well as how to create variables and call functions. These conceptual building blocks of a language act as a lingua franca irrespective of the specific language in question. The same is true within a language paradigm. Just like the semantics of classes and objects in an object-oriented language act as a lingua franca in the world of object-oriented programming, the function exclusively serves this purpose in a functional programming paradigm.

With just a few concepts the bulk of application design problems can be simply solved, particularly in data analysis. It is no secret that modeling data involves a lot of data processing. The steps involved typically include retrieving data, cleaning and normalizing data, persisting data for later use. And that’s just to prepare for doing the real work, which involves analyzing the data, creating and validating models, and finally running them on new data. A curiosity of data analysis is that mental effort is split roughly 20 to 80 between data processing and modeling, but program code is often the opposite, with data processing taking up the majority of the lines. This is due to the steps involved and the inherently messy nature of data versus the pure and ideal world of mathematics and models. As data moves between libraries, components, and systems, the formats and data structures are often incompatible. Making these disparate pieces of software interoperable requires ad hoc data transformation to fit all the pieces together.

In this chapter, we’ll see that first-class functions provide the necessary foundation to make it all possible. Higher-order functions provide the semantics for transforming data. The three primary constructions are iteration (map), recursion (fold), and set operations (filter). Closures complete the pic-
ture by providing semantics for conforming function interfaces. Functions are not generally compatible so it is typical that the interface to one model is not immediately compatible with another function. Closures act as the glue between the data structure returned by one function and the expected format of another. With this core set of semantic constructs it is unnecessary to learn additional patterns and frameworks, meaning more time can be spent modeling and less on the dirty work of data transformation.

1.1 First-Class Functions

Structuring computer programs often begins by dividing programs into data structures and functions \(^1\) that operate on the data. In an object-oriented programming (OOP) paradigm, data structures (objects) have associated methods that automatically passes the object as an operand to the function. In many OOP languages, functions can only exist as a part of a class. Despite the current popularity of this approach, this organizational structure is somewhat arbitrary. Consider that a Turing Machine operates on an infinite tape containing symbols that represent both instructions and data. The same is true at the hardware level, where data and instructions are ultimately both represented as a sequence of bits (data). Hence, at a fundamental level there is not much to distinguish functions from data. This indifference is also present in the lambda calculus, where \(\lambda\) consists of lambda terms that are either lambda abstractions (functions) or variables. Consequently, functional languages treat everything as data. When functions are treated like variables, they are referred to as first-class entities.\(^2\)

All functions are first-class in R.\(^3\) As a reminder, the syntax for function definition assigns a function to a variable. This is no different from assignment of a data structure to a variable.

**Example 1.1.1.** Let’s start by defining a univariate function that increments its argument. In mechanical terms, we are assigning a function to the variable named `increment`.

\[
\text{> increment <- function(x) x + 1}
\]

In Example 1.1.1 we’ve declared the variable `increment` and assigned a function as its value. This function can now be used like any other variable.

**Example 1.1.2.** To convince yourself further that functions are treated like any other variable, we can add functions as elements of data structures. We can then extract the element, assign it to another variable, and finally call it.

\(^1\) Or procedures, sub-routines, etc.
Functions as a lingua franca

> some.funs <- list(sum, increment)
> some.funs
[[1]]
function (..., na.rm = FALSE) .Primitive("sum")
[[2]]
function (x)
x + 1

> f <- some.funs[[2]]
> f(4)
[1] 5

Functions can also be passed as arguments to other functions. This is common practice in R, typically with one of the `apply` functions. `apply` is used to iteratively process each element in an `array`, `matrix`, or `data.frame`. In two dimensions, an element is meant to be a row or column of the table-like structure. The function passed to `apply` is sequentially applied to each element in the data structure.

Example 1.1.3. Let’s look at the classic `iris` dataset to illustrate how `apply` works. The signature of `apply` takes a data structure, the margin, which controls whether the iteration is along rows (columns), and a function that is applied to each row (column). Therefore, the function is treated as a value that is passed into `apply`. For this first example, we want to compute the mean for each attribute. This implies that each element is a column of the `iris` dataset, so the function `mean` is passed a vector in each iteration.

> apply(iris[,1:4], 2, mean)
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
5.843333     3.057333     3.758000     1.199333

The purpose of `apply` is to provide the machinery around iteration, which is a generalized operation. We can verify this by replacing `mean` with another statistic, like standard deviation.

> apply(iris[,1:4], 2, sd)
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
0.8280661     0.4358663     1.7652982     0.7622377

Since `apply` uses the same machinery in both cases, the `structure` of the result is the same irrespective of our choice of statistic. The `apply` function is an example of a map operation, which is one of three primary higher-order functions. Map operations will be discussed in detail in Chapter ??.

By using `apply`, the argument to the first-class function only needs to know how to process a single element instead of a set of elements. By separating the mechanics of iteration, the same function can be used for a single vector or multiple vectors without modification or ceremony. One condition is that
any first-class function being passed to `apply` must have the same interface, since `apply` only has a single implementation. R provides a mechanism via the ellipsis to handle this situation, although a more idiomatically consistent functional approach is to use a closure, which is discussed in Section 1.3.

1.2 Higher-order functions

Treating functions as variables is a nice feature, but their value truly shines when coupled with higher-order functions. In general these functions provide the machinery for transforming data in a repeatable way. Since data analysis involves many individual records having the same general structure (e.g. vectors or table-like structures), it is beneficial to divide the data processing into a function that is responsible for manipulating a single record at a time, and a function that is responsible for the iteration over the records. The first function is a first-class function passed to the second function, which is a higher-order function. This is the separation of concerns that we saw in the previous section with `mean` and `apply`, respectively. We’ll see in Chapter ?? that there are other types of machinery to manage alternate iterative processes.

**Definition 1.2.1.** A higher-order function is any function that takes a function as an operand, returns a function, or both.

If we didn’t use higher-order functions, what would the `iris` code look like? Generally it requires initializing some data structure that represents the result, iterating over the original data structure and storing the values in a loop.

```r
> y <- c()
> for (i in 1:4) {
+   y <- c(y, mean(iris[,i]))
+ }
> names(y) <- colnames(iris)[1:4]
> y
```

<table>
<thead>
<tr>
<th>Sepal.Length</th>
<th>Sepal.Width</th>
<th>Petal.Length</th>
<th>Petal.Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.843333</td>
<td>3.057333</td>
<td>3.758000</td>
<td>1.199333</td>
</tr>
</tbody>
</table>

While there is nothing conceptually wrong with an imperative approach, notice how the mechanics of the iteration must be implemented explicitly. This means getting dirty with indices and initial values. It also means comingle the scopes of two distinct operations. If we want to preserve the labels, that also must be done explicitly. All these additional steps add complexity and make code more error prone. \(^2\)

\(^2\)To hammer this point home, even in this simple example the author made a syntax error when first implementing it.
Functions as a lingua franca

AbstractStatistic <- setRefClass(
  Class="AbstractStatistic",
  fields=list(),
  methods=list(
    initialize=function(...) { callSuper(...) },
    execute=function(data, ...) {
      stop("Interface should not be called directly")
    }
  )
)

Mean <- setRefClass(
  Class="Mean",
  fields=list(),
  contains="AbstractStatistic",
  methods=list(
    initialize=function(...) { callSuper(...) },
    execute=function(data, ...) { mean(data, ...) }
  )
)

FIGURE 1.1: Generalization of iris statistics using object-oriented programming

and encapsulate general operations, and this applies equally to mathematical operations as well as to algorithmic operations. Functional programming gives us the tools to leverage both with ease.

Continuing with our hypothetical situation, suppose that the same generality achieved with apply and mean is desired. What options are available? A naive approach is to use a function with a string argument to generalize the statistic being used. In this case the whole loop is bundled up in a function and a large if-else expression or case statement is used to control flow.

function(data, statistic) {
  for (i in 1:ncol(data)) {
    if (statistic == 'mean')
      y <- c(y, mean(data[,i]))
    else if (statistic == 'sd')
      y <- c(y, sd(data[,i]))
    else
      ...
  }
}

This approach is clearly not generalized since the supported statistics are hard-coded in the function. A more viable approach is to use object-oriented
aggregate_iris <- function(data, statistic) {
  result <- c()
  for (i in 1:4) {
    result <- c(result, statistic$execute(data[,i]))
  }
  names(result) <- colnames(data)[1:4]
  result
}

FIGURE 1.2: Using the Strategy design pattern

Techniques. In this approach a class needs to be defined that manages the dispatching for each different statistic. Figure 1.1 illustrates a typical approach using ReferenceClasses. The implementation is based on the Strategy design pattern [], which codifies the behavior in an abstract class (or interface) followed by a concrete implementation for each specific statistic.

Applying the Strategy pattern to the iris dataset involves replacing the explicit call to mean with a call to the instance method execute as seen in Figure 1.2. As a matter of convenience, the loop is encapsulated inside a function.

The final result is obtained by first instantiating an instance of the Mean class and then passing it to our newly minted function.

> m <- Mean$new()
> aggregate_iris(iris, m)
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
   5.843333    3.057333    3.758000    1.199333

Now the function is general in the way we want, but at what cost did we achieve this? Without first-class functions a simple iteration over a dataset becomes quite complicated. Not only is there a lot of ceremony required to use the function, it is harder to understand what the purpose of the code is. In general it is best to avoid complexity unless there is a tangible benefit from it. A good example of this is adding complexity to improve the performance of a function. When there is no tangible benefit from complexity, you are essentially paying to make your life more difficult.

1.2.1 Functions that take functions as arguments

Not all higher-order functions manage the machinery of iteration. A common pattern is to create a higher-order function to support arbitrary implementations of a specific operation in the function. Some common examples are how NAs are handled in a function or to support different models. The advantage of using first-class functions is that the possibilities are infinite, so the author of a function does not have to guess at which implementations to provide.
Functions as a lingua franca

```r
classify_iris <- function(x) {
  x$Sepal.LW <- x$Sepal.Length / x$Sepal.Width
  x$Petal.LW <- x$Petal.Length / x$Petal.Width
  x$SP.Length <- x$Sepal.Length / x$Petal.Length
  x$SP.Width <- x$Sepal.Width / x$Petal.Width
  randomForest(Species ~ ., x)
}
```

FIGURE 1.3: Iris classification with added features

```r
classify_iris <- function(x, na.fn) {
  cols <- c('Sepal.Length', 'Sepal.Width',
            'Petal.Length', 'Petal.Width')
  x[,cols] <- apply(x[,cols], 2,
    function(z) ifelse(is.na(z), na.fn(z), z))
  x$Sepal.LW <- x$Sepal.Length / x$Sepal.Width
  x$Petal.LW <- x$Petal.Length / x$Petal.Width
  x$SP.Length <- x$Sepal.Length / x$Petal.Length
  x$SP.Width <- x$Sepal.Width / x$Petal.Width
  randomForest(Species ~ ., x)
}
```

FIGURE 1.4: Custom handling of NAs

Instead, a package can author focus on the *ideal* interface, knowing that a user of the package can use functional programming concepts to conform the data to the package interface.

Suppose we want to train a random forest to classify the iris dataset. The original dataset only has four features, so we will create a function that adds some more features and then executes the random forest, as in Figure 1.3. This function expects the standard Iris dataset and appends additional columns to the `data.frame` prior to calling the random forest. Now suppose that the dataset contains NAs. How should these be handled? The simplest solution is to use a scalar value and replace all NAs with this value. However, this approach is clearly limited in its functionality. What if we wanted to provide arbitrary handling of NA values? Then it is better to pass a function with a defined interface to handle this. An example of such a function is in Figure 1.4.

When calling the function, we need to decide what the `na.fn` function should do. A first approach is to create a function that computes the mean of the non-NA values. But first we need to modify the Iris dataset by adding some NAs. We'll create a new `data.frame` instead of modifying iris directly.

```r
> iris1 <- iris
```
This function randomly adds 10 NAs to each column of the dataset, which is sufficient for our purposes.

Computing the mean for each column should be as simple as calling `mean`. It would be nice to reference the function directly, but the default behavior is to return `NA` if any of the values are `NA`. Instead we need to wrap `mean` inside a function that calls `mean` and explicitly sets `na.rm=TRUE`.

```
> classify_iris(iris1, function(x) mean(x, na.rm=TRUE))
```

Call:
randomForest(formula = Species ~ ., data = x)
Type of random forest: classification
Number of trees: 500
No. of variables tried at each split: 2

OOB estimate of error rate: 4%
Confusion matrix:

<table>
<thead>
<tr>
<th></th>
<th>setosa</th>
<th>versicolor</th>
<th>virginica</th>
<th>class.error</th>
</tr>
</thead>
<tbody>
<tr>
<td>setosa</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>versicolor</td>
<td>0</td>
<td>47</td>
<td>3</td>
<td>0.06</td>
</tr>
<tr>
<td>virginica</td>
<td>0</td>
<td>3</td>
<td>47</td>
<td>0.06</td>
</tr>
</tbody>
</table>

By following this approach `classify_iris` is now a higher-order function. Removing the function implementation that handles NAs and instead adding it to the function signature points to the separation of concerns that was mentioned earlier. In essence, the function `classify_iris` becomes focused on data management, while the logic inherent in the model is isolated in the function argument. Continuing this pattern, it is possible to abstract the model call as well, supporting any arbitrary model. This version is implemented in Figure 1.5. Notice that we set the default model to the original `randomForest` function. This has the effect of preserving past behavior despite adding functionality, which is generally advisable when refactoring code.

Suppose we want to evaluate the performance of a support vector machine. This is as trivial as specifying the model parameter with the `ksvm` function reference.

```
> library(kernlab)
> classify_iris(iris1, function(x) mean(x, na.rm=TRUE), ksvm)
```

Using automatic sigma estimation (sigest) for RBF or laplace kernel Support Vector Machine object of class "ksvm"

SV type: C-svc (classification)
parameter : cost C = 1
Functions as a lingua franca

classify_iris <- function(x, na.fn, model=randomForest) {
  x[,cols] <- apply(x[,cols], 2, 
    function(z) ifelse(is.na(z), na.fn(z), z))
  x$Sepal.LW <- x$Sepal.Length / x$Sepal.Width
  x$Petal.LW <- x$Petal.Length / x$Petal.Width
  x$SP.Length <- x$Sepal.Length / x$Petal.Length
  x$SP.Width <- x$Sepal.Width / x$Petal.Width
  model(Species ~ ., x)
}

FIGURE 1.5: A generalized classification function for iris data

Gaussian Radial Basis kernel function.
Hyperparameter : sigma = 0.292382695546997
Number of Support Vectors : 61
Training error : 0.026667

Notice how our original function is now exclusively responsible for data management and wiring, while the actual model logic has been separated from this function. This separation of concerns is similar to how `apply` separates general data management machinery from specific application logic. Achieving an explicit separation of concerns is one of the key benefits of functional programming. Compare this to object-oriented programming where class hierarchies must be created to support the generalization we accomplished above with a simple change to the function signature.

1.2.2 Functions that return functions

Instead of calling a function directly, sometimes it is better to first call a function that returns a function and then call the resulting function. The rationale is that by having two functions, it is easier to understand the purpose of both via explicit separation of concerns. The outer function acts as a constructor of sorts, initializing certain values of the returned function. This keeps the interface of the returned function clean and concise. Often this is required to make two interfaces compatible.

Continuing the iris example from the previous section, what if you want to tune some parameters of the classification model? Notice that we’ve codified a de facto model interface in our classifier function: `function(formula, data)`. It isn’t possible to support all the tuning parameters for every individual model, as it would make the interface extremely cluttered, while also having finite
setup_svm <- function(...) {
    function(formula, data) {
        ksvm(formula, data, ...)
    }
}

FIGURE 1.6: A higher-order function used to configure a SVM

utility. This is similar to the hard-coding conundrum on page 5. Instead it’s better to write a function that knows how to call a model with our specific parameters, thus preserving the interface defined by `classify_iris`. This requires calling a function that returns a function with the correct signature. Hence the higher-order function is responsible for matching the function signatures. Let’s say we want to swap out the kernel in our SVM. The native function call is

\[
\texttt{ksvm(Species } \sim \texttt{., iris1, kernel='besseldot')}.
\]

Without modifying `classify_iris`, let’s create a new function `setup_svm` that knows how to configure our model, as shown in Figure 1.6. By using the ellipsis argument, our new function is generic enough that it supports any arbitrary parameter that we may want to tune. This function is then called like this.

\[
\texttt{model } \leftarrow \texttt{setup_svm(kernel='besseldot')}
\]

\[
\texttt{classify_iris(iris1, function(x) mean(x, na.rm=TRUE), model)}
\]

This same pattern happens often with the `apply` functions when a parameter must be set in a function. In fact, this is what we did when defining the function for handling `NA`s. The only difference is that we defined the function inline, so there was no need to create an explicit higher-order function. Any generic higher-order function is defining a de facto interface for its function argument. To ensure compatibility with this expected signature, a higher-order function can be used.

### 1.3 Closures

When a higher-order function returns a function, that function is typically a closure. What differentiates a closure from a basic function is that a closure has an associated external scope bound to the function. This means that variables can be referenced outside the function scope and accessed as immutable values. The significance is that the closure provides a way to track interstitial state strictly within the context of the function in question. In pure terms these variables are immutable, such that the values are guaranteed to be constant.
within the closure. This property is essential for deterministic behavior and local reasoning of a program.

**Example 1.3.1.** To see how a closure works let’s first define a simple function that references a variable in the global environment. This is not a recommended practice as it is unsafe, but for pedagogical purposes it is particularly illuminating.

```r
x <- -5
f <- function() {
  x <- abs(x)
  function(y) y - x
}
```

When calling this function, note that the original \( x \) is not modified.

```r
> g <- f()
> g(6)
[1] 1
> x
[1] -5
```

Thanks to lexical scoping, where the scope of a function is determined lexically from the source code [], closures are easy to create in R. This means that variables defined in an outer scope can be referenced within a function body. Doing so will make a copy of the variable in the closure environment. This implies that if the original variable changes value later in a program, it will not affect the closure. In the above example, changing \( x \) to another value will not affect the result of \( g(6) \).

**Example 1.3.2.** Avoiding the use of the global environment, let’s instead create a higher-order function that defines the same variable \( x \), which is referenced in the returned function.

```r
f <- function(x) {
  x <- abs(x)
  function(y) y - x
}
```

By debugging the closure, we can explicitly verify that the variables \( x \) and \( y \) are in separate environments.

```r
> g <- f(-5)
> debug(g)
> g(7)
  debugging in: g(7)
  debug: y - x
  Browse[2]> x
  [1] 5
  Browse[2]> ls()
  [1] "y"
```
1.3.1 Interface compatibility

We now know that the setup_svm function defined in the previous section returns a closure. The resultant function references a single variable outside of its scope, which happens to be the ellipsis argument. This is easily verifiable by manually inspecting the definition of the closure.

```r
> model
  function(formula, data) {
    ksvm(formula, data, ...)
  }
<environment: 0x7f985b8c1a38>
```

Notice that the ellipsis is present in the body of the function but the value is unspecified. To inspect its contents, a similar procedure as in Example 1.3.2 is required.

```r
> ls(envir=parent.env(environment(model)))
```

Any time a higher-order function specifies a function signature that is different from the signature of the function we want to pass to it, a closure is used to bridge the gap in signatures. The key is that the signature of the closure must always match the expected signature, while the higher-order function generating the closure can be arbitrary. Below is a simple algorithm for extending a function signature to become a higher-order function.

1. Add a first-class function argument to the signature
2. Replace explicit function call with argument
3. Create a new higher-order function to return a closure that calls the replaced function
4. Add arguments as necessary to higher-order function

This simple procedure works for any function that you want to turn into a higher-order function. This is the process followed when refactoring classify_iris to use an arbitrary model.

Let’s again pretend that we are not using functional programming. In this hypothetical scenario, we no longer have access to first-class functions. What are alternative ways to implement the same behavior? There are numerous approaches to this, but none of them are as simple and straightforward as using a first-class function.

One approach is to follow the approach of the standard optim function, where a method character argument is used to describe the optimization method. Actual dispatching is then performed explicitly via a conditional
Functions as a lingua franca

classify_iris <- function(x, method='rf') {
    if (method == "rf")
        randomForest(Species ~ ., x)
    else if (method == "svm")
        ksvm(Species ~ ., x)
}

block, which is the same naive approach we started with when we wanted to
generalize the use of a statistic on the iris data. Clearly this approach holds
constant the number of methods possible to use. In some circumstances where
the methods are truly finite this is acceptable. In other cases where the set of
methods are not known a priori it can be limiting.

One argument supporting a conditional block is that if each model expects
slightly different data, it might be easier to use the conditional block to control
the data transformations. This actually is the approach optim takes, which
unfortunately also shows how easy it is to create complicated code. Functions
provide explicit boundaries between blocks. This constraint forces a separation
of concerns, which makes it easier to reuse functions and modify them later.
In Chapter ?? we’ll see how to effectively swap out models while conforming
to their unique interfaces.

Rather than using an explicit conditional block one might use dynamic
function calls via do.call.

do.call(method, list(Species ~ ., data))

This approach supports any arbitrary function to be called, which is similar
to using a first-class function directly. The danger here is that a syntax error
will result in an execution error. Using a first-class function is safer since the
object is guaranteed to be callable.

Using ReferenceClasses provide a more traditionally object-oriented ap-
proach but still incurs much software design overhead to accomplish a simple
task. The strategy here is to create a class hierarchy to represent the different
types of models. Then a method is called to execute the model for the given
data. This design is similar to the design we used in Figure 1.1 for generalizing
the choice of statistic.

model <- RandomForestModel(data=iris1)
modelclassify(Species ~ .)

This approach is typical of an object-oriented paradigm. Notice how much
additional work is required to implement this style of programming. In general,
object-oriented design patterns stem from the need to create object structures
that emulate the behavior of first-class functions. Consider that with the in-
clusion of first-class functions in a language, the need for design patterns all
but disappears.
Modeling Data With Functional Programming In R

AbstractModel <- setRefClass('AbstractModel',
  fields=c("data"),
  methods=list(
    classify=function(formula) stop("not implemented")
  ))

SvmModel <- setRefClass('SvmModel',
  contains="AbstractModel",
  methods=list(
    classify=function(formula) ksvm(formula, data)
  ))

RandomForestModel <- setRefClass('RandomForestModel',
  contains="AbstractModel",
  methods=list(
    classify=function(formula) randomForest(formula, data)
  ))

FIGURE 1.7: A class hierarchy for classification models

1.3.2 State representation

In certain cases shared mutable state is appropriate to use, particularly for representing external resources. These resources are often singletons in the physical world (or in the operating system environment), so modeling them as a single shared object with state makes sense. Connections are an obvious example, where a resource is opened, read, and finally closed. Here a file descriptor represents the state of the file and must be managed accordingly.

Object-oriented paradigms are often heralded for their ability to manage state. In an object-oriented paradigm a class represents a generic file, and an instance of the class is a specific file. This file object can then be opened, read, and closed. The power of the object-oriented approach is that all resources, variables, and operations associated with the file are encapsulated within the class definition. The challenge is that each resource and method returns its own instances of other classes. Knowing when to stop modeling the class hierarchy is one of the hardest problems in designing object-oriented systems as one must balance reusability with ease of use. Highly granular class libraries are good for reuse, but it leads to exceptionally verbose implementations that are difficult to learn. In Java, there are distinct classes for files, connections, streams, and buffers. Loading a file in Java requires interacting with objects from each of these classes, which means understanding how a file system is modeled along with their individual APIs, in addition to the implicit state machines embedded within the class. An example of this are connections that must be closed after opening. When resources aren’t properly closed, it can lead to memory leaks as well as running out of operating system resources. Despite
Functions as a lingua franca

using <- function(resource, handler, exit=close) {
  tryCatch(handler(resource),
    error=stop, finally=function() exit(resource))
}

FIGURE 1.8: A resource management function

all this granularity, you still have to manually manage the actual resources
being modeled. The saving grace is that all of the machinery for managing
a resource can be encapsulated in a single class, which limits the hunt for
documentation. On the other hand, languages that favor monolithic classes
(like Objective-C) are also difficult to learn because so many permutations
exist for performing an operation that it isn’t immediately obvious which one
to use.

So the benefit of object-oriented programming comes at the cost of com-
plexity. Not surprisingly, functional programming provides a liberating alter-
native to the tyranny of all-encompassing class hierarchies. Rather than at-
tempts to optimize an interface for the most common use cases, functional
programming interfaces are restricted in quantity. Since closures are so easy to
create (and their resources managed efficiently), it is often trivial to conform
two interfaces together on an ad hoc basis. This approach preserves a simple
and clear interface for functions while avoiding the slippery slope of optimal
interface design.

In terms of state management, closures can provide the same encapsulation
as a class can. The key difference is that creating a closure does not require
a lot of ceremony and is therefore efficient in implementation. Closures can
be created ad hoc as an anonymous function or more formally as the return
value of a higher-order function. Any resources defined in the closure can be
automatically garbage collected once all references to the closure are gone. The
result is a cleaner code base since there are fewer formal type/class definitions.

A functional approach to managing resources involves, not surprisingly, a
higher-order function. We will implement a function that is inspired by the
with keyword in Python. A with statement automatically manages resources
within the scope of a block. When the end of the block is encountered or an
error is encountered, the specified resource is automatically closed. 3 Since R
defines with as a technique to access objects as environments, we’ll call our
version using.

The function is used like

z <- using(file(path), readLines)

The value of a function like this is that any errors in the handler will
automatically close the resource.

3In Python, with operates on a callable object that has a __enter__ and __exit__
function defined.
setup_using <- function(resource, exit=close) {
  function(handler, destroy=FALSE) {
    if (destroy) return(exit(resource))
    tryCatch(handler(resource),
      error=function(e) { exit(resource); stop(e) })
  }
}

FIGURE 1.9: Using a closure to manage external resources

z <- using(file(path), function(x) { log('a'); readLines(x) })

Example 1.3.3. Another scenario is managing graphical parameters. Sometimes a function needs to change these parameters to display a custom plot. A good citizen will ensure that the original parameters are restored once the function exits. A typical implementation looks like

plot_handler <- function(x) {
  opar <- par(mfrow=c(2,2), ...)
  on.exit(par(opar))
  # Do stuff
}

The use of on.exit is required to properly account for errors that may arise in the function. Without this inclusion, the parameters will not be restored properly if an error is encountered. This approach works well but is easily overlooked. The same can be accomplished with using.

using(par(mfrow=c(2,2), ...), plot_handler, par)

Notice how this approach cleanly separates the mechanics of managing the state of the graphics environment from the visualization code.

In the above cases no closure is required because the handler operation is effectively atomic. What if the resource must stay open for an indefinite period of time? Here a closure can be used to manage the resource. While the above technique is useful for a fixed set of operations, it doesn’t work well for arbitrary operations in disconnected control sequences. Taking a cue from Javascript, we can overload a function with multiple behaviors to achieve the desired behavior. Named parameters makes this a simple and safe exercise as seen in Figure 1.9. The general method is to define the default operation as the primary interface for the signature. Other operations are then controlled by optional arguments to the function.

Our new function using_fn is the second type of higher-order function

4The removal of the par lines in plot_handler is implied.
Functions as a lingua franca

setup_svm <- function(...) {
  errors <- c()
  function(formula, data, get.error=FALSE) {
    if (get.error) return(errors)
    result <- ksvm(formula, data, ...)
    errors <<- c(errors, result@error)
    result
  }
}

FIGURE 1.10: Evaluating the numerical stability of SVMs

since it is returning a function instead of having a function as an argument. Working with this function involves naming the returned function and calling this in lieu of using.

```r
> cat("line1\n", file="example.data")
> using.resource <- setup_using(file("example.data"))
> using.resource(readLines)
[1] "line1"

> cat("line2\n", file="example.data")
> using.resource(readLines)
[1] "line2"

> using.resource(destroy=TRUE)
> unlink("example.data")
```

The recurring theme of separation of concerns is yet again the main benefit. By clearly thinking about what is model logic versus general software machinery provides an opportunity to cleanly implement models according to the mathematical sequence of function composition. Once this distinction in code purpose is made, it also becomes clear that much of the data management machinery is general and can be easily reused at a level of sophistication that exceeds granular functions. This is because we have encoded a process workflow within a higher-order function and closure as opposed to a single operation.

1.3.3 Mutable state

In the previous section, the state being managed was static. Once a file resource is opened, the resultant connection doesn’t change state. Other situations have dynamic state that requires updating this state. Typically variables retained in a closure are immutable, but with the special <<- operator, it is possible to change the value of a variable.
Continuing the iris example, machine learning methods are often faced with the question of numerical stability. How do we know whether the solution from one iteration is representative of the model or is an outlier? Some algorithms have built-in stability tests [], whereas others require manual evaluation of the stability. We can answer this question by examining the distribution of the model results over multiple iterations.

To measure the classification error of the SVM we used earlier, we can modify the setup_svm function to track the errors over multiple iterations, as shown in Figure 1.10. By using a closure it is possible to preserve the previous function signature, allowing us to use this new function wherever the old one was used. Hence, we can perform as many iterations as we like in the same way as running any other model.

```r
> do.svm <- setup_svm()
> z <- sapply(1:500, function(x) do.svm(Species ~ ., iris))
```

As an added benefit, extracting the accumulated error is a repeatable process, so it is easy to work with the data. The histogram in Figure 1.11 is one way to view the classification error.
Functions as a lingua franca

seq.gen <- function(start)
{
    value <- start - 1
    function()
    {
        value <<- value + 1
        return(value)
    }
}

FIGURE 1.12: A simple generator function

> hist(do.svm(get.error=TRUE), main='SVM class.error',
+     xlab='Error')

The significance of this approach cannot be stressed enough. By using functional programming techniques, we’ve added specialized functionality for measuring error that preserves existing function signatures. This means that you can measure the error of a numerical method from any external package with little effort.

If we wanted to compare the performance of different kernels, this can be done with the current function.

> bessel.svm <- setup_svm(kernel='besseldot')
> tanh.svm <- setup_svm(kernel='tanhdot')

Since each closure manages its own state, it is easy to compare the error from each kernel. All this was accomplished with just a few lines of code. As a thought-experiment, compare this to what is required in a non-functional approach.

Some care does need to used with the <<- operator. This is due to the semantics involved: until a matching variable is found, the operator will continue to access enclosing environments until the global environment is found. At this point if no variable is found, one is created. Careless usage can therefore result in variables being created in the global environment.

Exercise 1.1. Rewrite setup_svm to be a general function that measures numerical stability for any model.

1.3.4 Generators

Building on the functionality of state management, closures can also be used to implement generator functions. Like all closures, generators have local state that is mutable but only accessible to the particular function. Generators are useful when a variable acts as a shared singleton, which requires explicit management of its internal state.

Example 1.3.4. A sequence generator is used for creating monotonically increasing IDs as seen in Figure 1.12. The returned closure does two things:
increment a counter and return its value. Thus a monotonic sequence is produced by calling the function successive times.

```r
> g <- seq.gen(5)
> g()
[1] 5
> g()
[1] 6
```

In general, generators are a convenient way to localize the side effects of a function. This abstraction also means that multiple instances of a generator can be created without worrying about namespace collisions. We’ll see in Chapter ?? how to use the generator concept for implementing finite state machines as well as Markov Chains for simulation.

## 1.4 Functions In Mathematics

The idea of first-class functions is not some radical idea spawned in the age of computing. Disciplines like traditional calculus actually makes extensive use of this concept. The derivative and integral both take functions as operands so conceptually these functions are being treated as data. It is also clear from the notation that a function is treated as a first-class entity. Take for example the polynomial function \( f(x) = ax^3 + bx + 4 \). When we take the derivative of this function the Liebniz notation hints at the concept: \( \frac{d}{dx} f \). Hence, \( f \) is a first-class function passed to the derivative operator, which just happens to be a higher-order function. Conceptually this is no different from writing the derivative as a function \( d(f) \equiv \frac{d}{dx} f \) for the univariate case.

First-class functions make an appearance in other parts of mathematics as well. Finding such cases often involves transforming operators into functions. This is a legal transformation, as we can prove that there is exactly one unique function that represents the operator.

**Theorem 1.4.1.** Given an operator \( \circ \), \( \exists \) exactly one function \( f : X \times X \rightarrow X \) such that \( f(x, y) = x \circ y, \forall x, y \in X \).

**Proof.** We use a proof by contradiction to show this is true. Let \( \circ \) be an operator \( \circ : X \times X \rightarrow X \). Define \( f_1(x, y) = x \circ y \). Suppose there is another function \( f_2 \neq f_1 \) where \( f_2(x, y) = x \circ y \). But \( x \circ y = f_1(x, y) \), so \( f_2 = f_1 \).

For the time being what is important is looking at the operand of the derivative. Here the function \( f \) is being transformed into the function \( f'(x) \). When writing functional programs it is useful to remember that this equivalence exists. In fact all operators in R are indeed functions. It is only the syntax that differentiates the two. However, any operator can be called using function notation.
Example 1.4.1.  
> `(4, 5) + (4, 5)`  
[1] 9

Example 1.4.2. The summation operator illustrates the equivalence between operators and functions. Suppose we want to take the sum of $f$ applied to each element of a vector. The expression is written mathematically as $\sum_i ax_i^3 - bx_i + 4$, which is really just fancy notation for function application. With some basic symbolic manipulation, we can illustrate this point. We’ll define a function $\text{sum}$ as follows.

$$
\text{sum}(\vec{x}, f) = f(x_1) + f(x_2) + \cdots + f(x_n) \\
= \sum_i f(x_i) \\
= \sum_i ax_i^3 - bx_i + 4
$$

This example shows that the summation operator is really just a function that takes a vector and a first-class function as arguments. We’ll see in Chapter ?? that $\sum$ and $\prod$ are examples of fold operations. The lesson here is that there shouldn’t be any bias in terms of using functions as operands to other functions.

Transforms are another class of mathematical entities that operate on functions. Consider the Laplace, Z, or Fourier transform. Each of these transforms takes an expression or function as an argument and returns a new function with change of variable. Hence, these transforms are a special type of higher-order function.

Example 1.4.3. Recall the definition of the Laplace transform, which is $\mathcal{L}\{f(t)\} = \int_0^\infty e^{-st}f(t)\,dt$. The notation is clearly indicates that $f(t)$ is the operand to the function $\mathcal{L}$.

When thinking of a transform $y = f(x)$ we often discuss the inverse $g(y)$ as well, which has the property of reversing the operation of $f$. In math terms we have $x = g(f(x))$, for all $x$ in the domain of $f$. This is true of transforms and is reflected in the relationship between the derivative and the integral. While not all programming functions have inverses, thinking about functions as being analytic or as transforms helps to prime your thinking. Leveraging the tools of mathematical analysis for the act of model implementation facilitates reasoning about the program code. As we progress further in the book, numerous examples of this will be highlighted.
1.5 A lambda calculus primer

The astute reader will likely notice the gradual arc towards mathematical reasoning in this chapter. The goal is to highlight the shared semantic structures in the notation of mathematics and functional programming. This overlap yields remarkable clarity in thinking about models and data. Functional programming is possible thanks to the conceptual foundation laid by the lambda calculus. Invented by Alonso Church, the lambda calculus defined computable functions to answer the so-called Entscheidungsproblem [2].

As an outgrowth of this task, much of the mathematical landscape like numbers and algebra were defined using this system. Defining the whole of mathematics is out of scope for this book; what we care about is the ability to define higher-order functions and closures. We also need a notation for anonymous functions, which the lambda calculus provides us. Formalizing these concepts will enable us to perform symbolic transformations so that we can fully reason about our functional programs. To start we establish some basics regarding the lambda calculus. Our focus will be the untyped lambda calculus as it is readily compatible with a dynamically typed language like R. In the untyped lambda calculus only variables \( v_1, v_2, \ldots \), the symbols \( \lambda \) and \( . \), and parentheses are allowed. The set of all lambda expressions is further defined inductively [2].

**Definition 1.5.1.** The set of all lambda expressions \( \Lambda \) is defined by

- (a) If \( x \) is a variable, then \( x \in \Lambda \).
- (b) If \( x \) is a variable and \( M \in \Lambda \), then \( \lambda x.M \in \Lambda \).
- (c) If \( M, N \in \Lambda \), then \( (MNN) \in \Lambda \).

This definition tells us that variables, functions, and the result of functions are all lambda terms. Typically uppercase letters are used to denote lambda terms while lowercase letters represent simple variables. So long as the mathematical constructions we create satisfy this definition, then we can leverage the lambda calculus in our analysis.

In the lambda calculus all functions are anonymous and first-class. Anonymous functions are therefore synonymous with lambda abstractions. A named function is thus nothing more than a lambda abstraction bound to a variable. These are denoted as in conventional mathematics. Hence \( f(x) \equiv f = \lambda x \). This equivalence can be extended to any function with an arbitrary number of arguments. For function application we note that \( (\lambda x.M)[x := N] \equiv f(N) \), where \( f(x) = M \).

In terms of symbolic notation, equality of expressions is denoted by \( = \). For recursive functions, it can be difficult to distinguish between symbolic equality of an expression and equality based on a recursive application of a function. In these cases \( \rightarrow \) is used instead.
1.5.1 Reducible expressions

Lambda terms can be transformed under certain conditions, referred to as either a conversion or a reduction. As one might expect a conversion changes the syntax but not the form of an expression. The most fundamental conversion is an $\alpha$-conversion, which is commonly known as a change of variable. Any reference to a given variable can be replaced with a different variable without changing the meaning of the expression. For example $\lambda x.x \ast 2 = \lambda y.y \ast 2$. Reduction is the process of simplifying an expression using rewrite rules. The goal is to achieve a so-called normal form that cannot be reduced further. Applying arithmetic operations can be considered reductions since repeated application of the operations eventually yields a scalar value, which is a terminal value.

Most algebraic operations can be viewed as a conversion or reduction. Consider a polynomial that requires factorization in order to simplify it as

$$f(x) = \frac{x^2 + 3x + 10}{x - 2} = \frac{(x - 2)(x + 5)}{x - 2} = x + 5.$$  

We can think of the first step as a conversion since neither form is clearly simpler than the other. Hence these two forms could be interchanged for an indeterminate number of times without ever arriving at a normal form. Once the term $x - 2$ is factored out, then it is clear that a reduction operation can take place, eliminating this term from both the numerator and denominator.

Operators like the factorial are also governed by rewrite rules. For example $5! = 5 \ast 4! = 20 \ast 3! = 60 \ast 2! = 120$. We can look at the factorial as either an operator or a function. When viewed as a function we must describe the mechanics of function abstraction, which provides a syntax for defining a function. Furthermore we must consider function application and the role of variable substitution within an expression.

Function application is governed by $\beta$-reduction, which tells us how to apply a function $M$ to a lambda term $N$, or $MN$. If $M$ is a function of variable $x$, then this application is denoted $MN = M[x := N]$. Suppose that $M = \lambda x.X$, then $(\lambda x.X)N = (\lambda x.X)[x := N] = X[x := N]$. The latter expression can be read as $X$ evaluated with $x$ replaced with $N$. Referring back to the factorial function, we can define this as a lambda abstraction $\lambda x.x!$ and apply it to the parameter $5$ as $(\lambda x.x!)[x := 5] = x![x := 5] = 5! = 120$.

The final conversion is known as $\eta$-conversion and is often characterized in terms of extentionality. I tend to think of $\eta$-conversion more as a proclamation of independence between lambda terms. In other words a function application has no effect on an embedded lambda term if there is no dependency on the argument. Recognizing the applicability of $\eta$-conversion can often lead to greater modularity and simplification of an algorithm.
Definition 1.5.2. Given \( \lambda x. M x \) where \( M \) is a lambda abstraction, if \( x \) is not free in \( M \) then the \( \eta \)-conversion of the lambda abstraction is \( \lambda x. M x \leftrightarrow_\eta M \).

The standard conversions and reductions provide mechanisms to reduce lambda terms into normal forms. Sometimes it is useful to go in the opposite direction and add structure to a lambda term. This is analogous to unconventional factorizations of polynomials to achieve a particular goal.

Proposition 1.5.3. An equivalent higher-order function can be constructed from any existing function. This step is an intermediate one on the way to creating a closure as discussed in Section 1.3.

\[
\lambda w.X = \lambda v.(\lambda w.X)[w := v]
\]

Proof. Given \( \lambda w.X \) and \( n \in \Lambda \). The left-hand side reduces by standard function application to \( (\lambda w.X)n = X[w := n] \). The right-hand side reduces to

\[
\lambda v.(\lambda w.X)[w := v] = \lambda v.X[w := v] \\
(\lambda v.X[w := v])[v := n] = X[w := n].
\]

Example 1.5.1. Let \( f = \lambda x.x + 1 \). Then

\[
f = \lambda x.f(x) \\
= \text{function}(x) f(x)
\]

The last line shows the equivalent syntax in R. Let’s look at a concrete example in the interpreter.

\[
> f <- \text{function}(x) x + 1 \\
> f(5) == (\text{function}(x) f(x))(5) \\
[1] \text{TRUE}
\]

This example shows the equivalence between the two constructions for a single value. The lambda calculus gives us the tools to prove that the equivalence holds for all values of \( x \).

Standard mathematical notation can be included as lambda expressions, since numbers are variables and operators are functions. When including function application using traditional notation, care must be taken with the variable naming.

Example 1.5.2. In example ?? we used the same variable \( x \) for both functions. To avoid confusion, it is wise to apply an \( \eta \)-conversion to one of the function definitions.

\[
f = \lambda y. f(y) \\
= \text{function}(y) f(y)
\]
The lambda calculus also supports multivariate functions via Currying [2]. Additional arguments are appended after the \( \lambda \) symbol as \( \lambda xy.z.X \).

**Example 1.5.3.** The \( \text{ksvm} \) function has multiple arguments, but let’s assume that it is defined as \( \text{ksvm} = \lambda \text{formula} \text{ data}.M \). Then

\[
\text{ksvm} = \lambda \text{formula} \text{ data.ksvm(formula, data)} \\
= \text{function(formula, data) ksvm(formula, data)}
\]

A closure can now be constructed by applying the proposition to a multivariate function. This means that at a fundamental level, we can create a closure from an existing function and be certain that its behavior is unchanged.

**Example 1.5.4.** As a shorthand I will often denote a set of function arguments as a sequence. Hence for \( W = \langle x, y, z \rangle \), \( \lambda xyz.X = \lambda W.X \). To illustrate the creation of a closure along with this point, let’s rewrite a version of \( \text{setup_svm} \) in lambda calculus notation.

\[
\text{setup_svm} = \lambda W.\lambda \text{formula} \text{ data.ksvm(formula, data, W)} \\
= \text{function(...) function(formula, data) ksvm(formula, data, ...)}
\]

There is no requirement that a lambda abstraction must only reference variables specified in the function signature. Those that are present in the argument list are known as **bound**, while those not present are **free**. The relationship between free and bound variables within lambda abstractions form an important basis for understanding how to transform functional programs into equivalent forms. We will explore this in depth in Chapter ??.

\[ \]

### 1.6 Church numerals

To understand how functions can be treated as data, the Church numerals provide a good example of how this works in practice. Church numerals represent a technique for encoding the natural numbers based on the lambda calculus. The insight is that any structure or process that is countable can be mapped to the cardinal numbers. In traditional mathematics, set theory is often used to show how the fundamental entities of mathematics can be used to prove the existence of natural numbers. Doing so reduces the axioms that mathematics must rely on. Once natural numbers are defined it is easy to derive the integers followed by the rational numbers.

As a product of the lambda calculus, Church numerals are simply functions. Yet these functions can be operated on just like the natural numbers.

**Definition 1.6.1.** Church numerals are based on the definition of function composition, which is defined inductively [2]. Let \( F, M \in \Lambda \) and \( n \in \mathbb{N} \). Then
TO_NAT <- function(x) x + 1

FIGURE 1.13: Mapping Church numerals to natural numbers

\[ F^n(M) = M \text{ and } F^{n+1}(M) = F(F^n(M)) \]. The Church numerals are then defined as \( c_n = \lambda f. f^n(x) \). For example,

\[
\begin{align*}
  c_0 &\equiv \lambda f. \lambda x. x \\
  c_1 &\equiv \lambda f. \lambda x. f(x) \\
  c_3 &\equiv \lambda f. \lambda x. f(f(f(x)))
\end{align*}
\]

The syntax in R is equivalent despite the nominal syntactic differences.

C0 <- function(f) function(x) x
C1 <- function(f) function(x) f(x)
C3 <- function(f) function(x) f(f(f(x)))

By definition these functions represent scalar values and thus can be considered data. This means that the functions can be operands to other functions and indeed this is the case with addition. The addition operator is derived from the successor function, \( \text{SUCC} \), which simply increments a Church numeral.

\[
\text{SUCC} \text{ c}_2 = \text{SUCC}[n := c_2] 
\]

\[
= (\lambda n. \lambda f. \lambda x. f((n(f)x)))[n := 2] 
= \lambda f. \lambda x. f((\lambda g. \lambda y. g(g(y)))[g := f])x 
= \lambda f. \lambda x. f((\lambda y. f(f(y)))[y := x]) 
= \lambda f. \lambda x. f(f(f(x))) 
= c_3
\]

In the case of Church numerals, both the numerals and the operators are higher-order functions. One would think that in a computing environment, it would be easier to verify the operation performed by the \( \text{SUCC} \) function. Due to lazy evaluation, it is actually somewhat opaque and requires an additional step to verify the complete computation. Let’s see what happens in the interpreter when we apply \( c_2 \) to the \( \text{SUCC} \) function.

\[
> \text{SUCC}(c_2) \\
\text{function(f) function(x) f(n(f)(x))}
\]

<environment: 0x7fe521d88740>

\[5\]Note that an \( \alpha \)-conversion is applied to 2 for clarity’s sake.
Functions as a lingua franca

PLUS <- function(m) function(n) m(SUCC)(n)

FIGURE 1.14: Addition for Church numerals

Since the return value is an unevaluated function, the arguments of the SUCC function are unevaluated. This ties into the idea of a closure, where the bound variables are unevaluated. Hence to see a value that is meaningful to humans requires evaluating the function completely. To do so requires creating a function to map the function composition to the natural numbers, as in Figure 1.13. Then to verify the value of the Church numeral, simply call every function with the appropriate argument.

> SUCC(C2)(TO_NAT)(0)
[1] 3

Finishing up, we can now define addition in terms of the successor function, as seen in Figure 1.14. The mechanics of the operation can be rather cryptic, so let’s break down how it works. Let’s evaluate PLUS c2 c3. The equivalent definition of PLUS is PLUS = \( \lambda m \ n \ m(SUCC)\). Recall that the definition of SUCC = \( \lambda n \ f \ x. f(n f x) \). The first part of the definition applies SUCC to \( c2 \), which gives

\[
c2 \ SUCC = (\lambda f \ x. f^2(x))[f := SUCC]
= (\lambda f \ x. f(f(x)))[f := SUCC]
= \lambda x. SUCC(SUCC(x)).
\]

Now apply \( c3 \) to this intermediate result, yielding

\[
(\lambda x. SUCC(SUCC(x))) \ c3 = SUCC(SUCC(c3)) = c5.
\]

In R, we can evaluate the sum using the same technique as before.

> PLUS(C2)(C3)(TO_NAT)(0)
[1] 5

The PLUS function illustrates how a Church numeral can be used as both data and as a function within the same equation. Initially \( c2 \) is used as a function that operates on SUCC. Then \( c3 \) is used as an argument to the resulting function, which yields a function representing a value. The value of Church numerals is that it provides a framework for thinking about function composition. We will revisit them later in terms of the fold concept.
1.7 Summary

The benefits of functional programming are legion, and this chapter highlighted many of these benefits. The primary theme surrounded the idea that functions can be treated like any another piece of data. Alternatively, data can represent both values and functions. We covered the core concepts of functional programming from first-class functions, to higher-order functions, to closures. This simple toolkit can be applied to virtually any situation offering a clean separation of concerns between model logic, data management logic, and application logic. The end result is a modular program with a clear delineation between reusable pieces of data logic and model-specific ad hoc pieces.

We also explored the mathematical connection with functional programming concepts, which will facilitate model development in subsequent chapters. The brief introduction to the lambda calculus provides a formal framework for understanding function transforms within code, which can simplify model implementation as well as provide insights into the model itself.
Vector Mechanics

Much of the elegance of R stems from the interplay between functional programming semantics and native handling of vectors. This unique combination makes R unrivaled in its ease of use for data analysis. We discussed the importance of functional programming in the previous chapter. In this chapter we explore vectors and what it means for vectors to be a fundamental data type. On the surface this simply means that the semantics of vectors are built into the core of the language. This is similar to how objects are fundamental to an object-oriented language, where every variable is an object [7]. In R, all primitives are vectors, meaning that any scalar value is really a vector of length one. This implies that vector operations can be performed on scalars without issue.

Coming from other languages, where scalars are primitives and arrays are separate data types, such a concept can seem alien. Yet this approach is very powerful in its expressiveness and again can be traced to its mathematical roots. In abstract algebra, the operators $+$ and $\times$ are polymorphic over rings, in the sense that they operate consistently over any ring. This is how we naturally think about addition. Regardless of whether the operands are scalars or vectors, we expect addition to operate in a semantically consistent way.

Recall that vector addition is performed when the dimensions of the operands are the same. This rule applies to vectors of arbitrary length. By definition these operations are performed on an element-wise basis. Given

$\mathbf{x} = \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$ and $\mathbf{y} = \begin{bmatrix} 4 \\ 4 \\ 8 \end{bmatrix}$, the sum is naturally $\mathbf{x} + \mathbf{y} = \begin{bmatrix} 7 \\ 9 \\ 15 \end{bmatrix}$. When working with matrices, we expect addition to again operate in this semantically consistent manner. R takes this concept and generalizes it to support arbitrary operators and functions.

The challenge faced by model builders is that mathematics is often silent about the implementation of an operation. Mathematical symbols and operators convey meaning regarding what an operator does but not how it does it. Even something as trivial as $\sqrt{x}$ can be articulated simply despite its implementation being difficult to describe. This dichotomy between what an operation does and how to compute it becomes even more pronounced with matrix operations, where concepts like the inverse or eigenvalues are again easy to describe but require much effort to explain how they are computed.
This emphasis on what an operation does but not how is the essence of a declarative programming style.

**Definition 2.0.1.** *Declarative programming* structures programs based on the logic of a computation without prescribing control flow. [?]

Declarative notation offers numerous benefits. By focusing on what an operation does, its fundamental behavior and properties can be communicated concisely. The details of implementation can obfuscate the intention of a proof or algorithm, which detracts from the meaning and ultimately the elegance surrounding an operation. The common theme is that mathematics, functional programming, and native support of vectors all promote a declarative style. In contrast, many programming languages are imperative in nature. Fundamental operators like addition only work for a handful of primitive data types, such as integers, doubles, and sometimes characters. Adding two vectors together therefore requires explicitly looping through both data structures to access each individual primitive element. This makes perfect sense when thinking about data structures from a hardware perspective but less so from a software or mathematical perspective. These programs have an imperative style, which reflects the underlying semantics of a Turing Machine.

**Definition 2.0.2.** *Imperative programming* structures programs based on a sequence of statements that manipulate program state. [?]

Imperative algorithms result in very explicit statements and program structure. This is borne out of necessity since the program state is managed explicitly. Hence a conceptually simple operation like vector addition all of a sudden requires initializing variables and incrementing indices to access individual elements of vectors. As an example, let \( \mathbf{x} <- c(3,5,7) \) and \( \mathbf{y} <- c(4,4,8) \). To compute the sum a new vector \( \mathbf{z} \) must be initialized, followed by assignment of each individual element:

```r
z <- c()
for (i in 1:length(x))
  z[i] <- x[i] + y[i]
```

By its very nature, data analysis is full of vectors and matrix-like structures. Imagine having to operate on vectors like this any time an ad hoc analysis is desired. The verbosity and tedium required rivals that of manually computing the inverse of a matrix. Surely there is a more efficient way to work with these data structures. Indeed, experienced R users would never sum two variables as shown above. They would instead take advantage of **vectorization** to perform the operation as a single statement:

```r
z <- x + y
```

Again, the key is that since all primitive types are vectors in R, the polymorphic properties of mathematical operators are preserved over scalars, vectors, and matrices. When functions (operators) operate on vectors and adhere to
specific properties, they are called vectorized functions (operators). In situations where you have to write your own operators and functions, we’ll see that higher-order functions provide the necessary semantic foundation that resembles this concept of vectorization. Some approaches for iterating over vectors are so common that they form a canonical set. The canonical higher-order functions include map, fold, and filter,

ch is the essence of Chapters 3 and 4, respectively. We won’t discuss the filter operation much since it is functionally equivalent to set comprehensions.

One might argue that this declarative style is nothing more than syntactic sugar. In a language like R that is ultimately built atop C, this holds more than an ounce of truth. Yet through the use of functional programming concepts, the declarative style goes beyond syntax, shaping not only the program structure but also how one thinks about the relationship between the code and the underlying mathematics. This chapter develops the mathematical properties of vectors. The motivation for a formal approach is to leverage the duality between programs and proofs. By focusing explicitly on revealing this duality, something remarkable happens: translating mathematical ideas into algorithms becomes a direct mapping. It becomes almost trivial to implement mathematical ideas, and we see this in much of R already. For instance, the covariance between two random variables $x$ and $y$ is a one line expression:

$\frac{(x - \text{mean}(x)) \%*% (y - \text{mean}(y))}{\text{length}(x)-1}$

As an added benefit, it becomes easy to reason about the program code, which leads to fewer implementation errors. It can also lead to insights into the mathematical transformations. Achieving this nirvana requires codifying the syntactic equivalences between R and mathematical notation. Since functional programming and native vectors facilitate declarative style, such a mapping is natural. The first step in this journey is analyzing the vector.

2.1 Vectors as a polymorphic data type

Until now we’ve talked about vectors without really defining what they are. As a data structure, vectors can represent numerous mathematical concepts. Beyond simply their mathematical namesake, the vector type can also represent $n$-tuples (sequences), sets, random variables, etc. Distinguishing between one mathematical type and another is a matter of context and also convention within a program. Irrespective of the mathematical type being modeled, vectors as a data structure are obviously governed by a single set of properties and rules. Properties like atomicity, and rules for concatenation and coercion are all important considerations when modeling a mathematical type with a vector. These implementation details of the data structure are important as they put specific constraints on the mathematical structures being modeled. For example, elements of a vector must all be of the same atomic type. The set
of atomic types is denoted $T$, and any vectors created with elements not in $T$ will result in a list. The use of a single data type to represent numerous mathematical entities also affords us a certain flexibility when working with data. At our convenience, it is possible to mentally switch the mathematical context and apply set operations as necessary to a structure that heretofore was considered a sequence. These transformations are legal so long as we understand the implications of the data structure for each mathematical entity.

A natural question arises from this polymorphism: what should the vector data type represent by default? In every day modeling, it is likely unnecessary to make a stance, but for the sake of notational consistency, it is a worthwhile question. Taking a pragmatic approach, an ordered $n$-tuple, or sequence, is conceptually similar and makes a good candidate. Sequences provide useful semantics for extracting elements from them, which is natural in data analysis. They also abide by the structural rules of the vector data type, as elements of a sequence form a domain $T^n$, where $T \in T$.

**Definition 2.1.1.** A vector is an ordered collection of elements equivalent to a finite sequence. A vector $x$ with $n$ elements is denoted as $e(x_1, x_2, ..., x_n) \equiv (x_1, x_2, ..., x_n)$. Vectors are governed by the following rules:

(a) Elements of a vector are indexed by a subset of the positive integers $N_n = 1, 2, ..., n$.

(b) For vector $x$ and $k \in N_n$, $x[k] \equiv x_k$ is the $k$-th element of $x$.

(c) The length of a vector is its cardinality and is denoted $\text{length}(x) \equiv |x|$.

(d) $x_k \in T$, $\forall x_k \in x$ where $T \in T$.

(e) Two vectors $x, y$ where $|x| = n = |y|$ are equal if and only if $x_k = y_k \forall k \in N_n$.

**Example 2.1.1.** Definition 2.1.1 provides the foundation for working with vectors. A brief example illustrates how the syntactic equivalences work. Given $x = (3, 5, 7)$, the first element is $x_1 = 3 = x[1]$. The overall length is $|x| = 3 = \text{length}(x)$.

**Example 2.1.2.** As described in Definition 2.1.1e, comparison of vectors is an element-wise operation. This is not limited to equality and is generally true of the relations defined in R. Hence to determine whether two vectors are equal requires comparing their individual elements. The equality operator is vectorized, so testing vector equality results in a logical of the same length as the operands. Let $x = (2, 3, 4)$ and $y = 2 : 4$. Then

```r
> x == y
[1] TRUE TRUE TRUE
```

To determine if the vectors are equal as opposed to the individual elements thus requires the use of another vectorized function `all`, which is equivalent to $\wedge$ in logic. The equivalent mathematical expression is $\bigwedge_i x_i = y_i$, which translates to the following in R:
2.1.1 Vector construction

Like a sequence, a vector is created by a function defined on \( \mathbb{N}_n \). The simplest vectors can be created explicitly using the concatenation function \( c \).

\[
> x <- c(2, 3, 4, 5)
> x
[1] 2 3 4 5
\]

Vectors can also be created using the sequence function, which is defined as

\[
\text{seq}(i, j, \text{by}=m) \equiv \{i, i + m, i + 2m, ..., j\}, \text{where } m \in \mathbb{R}. \]

If the sequence is an integer range, then \( i:j \equiv \text{seq}(i, j, 1) \) can be used as a shorthand. As we will see later, this notation is the same as the shorthand for extracting subsequences.

**Example 2.1.3.** In short, there are three ways to create integer sequences. The least flexible but the most concise is the \( : \) notation. For static sequences, using the concatenation function explicitly is simplest but can be tedious, while \text{seq} provides a balance of flexibility and ease of use.

\[
> \text{seq}(2, 5) == 2:5
[1] \text{TRUE TRUE TRUE TRUE}

> 2:5 == c(2, 3, 4, 5)
[1] \text{TRUE TRUE TRUE TRUE}
\]

**Example 2.1.4.** One thing to consider when using the \( : \) operator is that its precedence is very high. Hence certain constructions will result in behavior that is unexpected. This happens a lot when the end of the sequence is defined by a variable. Suppose we want to specify indices from 2:4 given a vector of length 5.

\[
> x <- \text{rnorm}(5)
> 2:length(x)-1
[1] 1 2 3 4
\]

Due to precedence rules, \( 2:n \) is evaluated first and then 1 subtracted from the resulting vector \(^1\). What we want instead is the sequence 2:4, which requires explicit parentheses.

\[
> 2:(\text{length}(x)-1)
[1] 2 3 4
\]

\(^1\)The precedence rules of R can often be surprising, such as \(-2^2 == -4\).
Traditional mathematical vectors are simply ordered $n$-tuples. When vectors need to explicitly represent traditional vectors, either tuple or matrix notation is used. For example, $\vec{x} = [3, 5, 7] \equiv \begin{bmatrix} 3 \\ 5 \\ 7 \end{bmatrix}$. In either construction, we'll refer to mathematical vectors exclusively via square brackets. These vectors are by default column vectors, which can be confusing to newcomers, since the printed output appears horizontal.

```r
> c(3, 5, 7)
[1] 3 5 7
```

However, performing a transpose on a vector will confirm that the original was indeed a column vector.

```r
> t(c(3, 5, 7))
[,1] [,2] [,3]
[,1] 3 5 7
```

### 2.1.2 Scalars

Since all atomic types are vectors, we can surmise that scalars are simply vectors of length one. From this perspective, all vector operations equally apply to scalars.

**Definition 2.1.2.** A vector $x$ is a *scalar* if and only if $|x| = 1$. It follows that $x = x_1$.

Most properties are semantically consistent when applied to scalars, while some unique properties also emerge. The most noteworthy is that scalars are invariant under indexing. Mathematically this amounts to saying that a set containing a single element is equal to the element itself, which is non-sensical. Yet this is perfectly legal in R and is a valuable contribution to the syntactic efficiency of the language.

**Example 2.1.5.** It’s easier to see the invariance over indexing by plugging some values into the interpreter.

```r
> x <- 3  # scalar
> x == x[1]
[1] TRUE
```

Compare this to a non-scalar, which results in a vector.

```r
> y <- c(3, 5, 7)  # non-scalar
> y == y[1]
[1] TRUE FALSE FALSE
```

---

$^2$Except possibly for an infinite recursive set.
In this latter comparison, each element of \( y \) is being compared to \( y_1 \), which is why the result is a vector.

The moral is that scalars in a vectorized language can yield idiosyncratic behavior that must be accounted for. Another such complication involves logical operators. Newcomers are typically confused between \& (1) and && (11). The difference is that the single character form is vectorized while the dual version is not.

**Example 2.1.6.** Let’s compare two common sequences. Define \( x = (1, 2, 3, 5, 7, 11, 13, 17, 19) \) and \( y = (1, 1, 2, 3, 5, 8, 13, 21, 34) \) where \(|x| = n = |y|\). When is \( y \) greater than \( x \)? Set builder notation describes the result as \( \{y_i | y_i > x_i, \forall i \in \mathbb{N}_n\} \). In this case, we only care about the comparison, as opposed to the set of elements that satisfy the condition.

\[ > y > x \]

\[ [1] \text{FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE} \]

The result illustrates that the conditional operators are vectorized as claimed earlier. Let’s also look at the occurrences when \( y \) is even.

\[ > y \%\% 2 == 0 \]

\[ [1] \text{FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE} \]

Now if we want to find the elements that satisfy both conditions, we use the single \& operator, since this is the one that is vectorized.

\[ > y \%\% 2 == 0 \& y > x \]

\[ [1] \text{FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE} \]

If instead we use the && operator, a scalar is returned. This is because the double operator is designed for conditional blocks that traditionally operate on scalar boolean values. When vectors are passed to these operators, only the first result is used, which can be surprising.

\[ > y \%\% 2 == 0 \&\& y > x \]

\[ [1] \text{FALSE} \]

Understanding the cardinality of operands and the cardinality expectations of an operation is central to writing proper code in R. We’ll see in Chapter 3 that a certain class of vectorized functions preserve cardinality while Chapter ?? explores another class that reduces the cardinality to one.

### 2.1.3 Atomic types

Earlier in the chapter, we casually mentioned the set of atomic types, known as T. Certain types are known as atomic because the data are stored in contiguous cells of memory. This arrangement is for efficiency reasons and results in subtle differences in behavior from the non-atomic types.
Definition 2.1.3. Let \( T = \{ \text{logical, integer, double, complex, character, raw} \} \) be the set of atomic types. [?]

Example 2.1.7. As stated in Definition 2.1.1, vectors comprise elements of a single atomic type. Some valid constructions of vectors are presented below.

\[
\begin{align*}
&> c(3, 5, 8) \\
&[1] 3 5 8 \\
&> c("a", "b", "aa") \\
&[1] "a" "b" "aa" \\
&> c(TRUE, NA, FALSE, TRUE) \\
&[1] TRUE NA FALSE TRUE
\end{align*}
\]

Note that we are careful in our use of the term atomic versus primitive. The set of primitive types in R contains the atomic types and includes other types such as environment, closure, and list. \(^3\) We will refer to some of these primitive types, but most of the discussion will be centered around the atomic types. Furthermore, notice that matrix is not an atomic type, let alone a type at all. The reason is that matrices are simply special vectors with an associated class that distinguishes them as matrices instead of vectors. The same is true of factors, which are integers having a class attribute of factor and other metadata. Hence, the type of a factor is still integer.

Example 2.1.8. Constructing a vector with values that are not of the atomic types will result in a list.

\[
\begin{align*}
&> typeof(c(list(1,1), 3, 5, 8)) \\
&[1] "list" \\
&> typeof(c(3, 5, function(x) x)) \\
&[1] "list"
\end{align*}
\]

2.1.4 Coercion

The previous example showed that concatenation will change its output type depending on the input types. Similarly, if a vector is created with values of different types, the values will be coerced into a type that is compatible with all elements. This coercion process ensures that the vector has at most one atomic type. Figure 2.1 displays a partial compatibility hierarchy between types. What this means is that with sufficient variation of types in the input, the output type will likely be character.

Example 2.1.9. Creating a vector from an integer, a double, and a logical results in a double since integers and logicals can be represented as doubles.

\(^3\)See ?typeof for more details.
character
   |
complex Date
   \ / 
   double
   |
integer
   / \ 
factor logical

FIGURE 2.1: Partial type coercion hierarchy for concatenation

> c(3, 4.5, FALSE)
[1] 3.0 4.5 0.0

However, mixing a number of types with a character results in a character vector.

> c(6, "a", TRUE, 5+2i)
[1] "6" "a" "TRUE" "5+2i"

The coercion process occurs even when the input does not appear to be a primitive type. The key is to look at the type and not the class of a variable. A variable's class is user-modifiable and determines the representation of a value. On the other hand the type is not user-modifiable and determines the actual storage mode of the variable. Consequently, vectors and matrices support only a single type, whereas lists (and structures derived from them) support one type per element.

**Example 2.1.10.** A Date object is actually a double with a class of Date. In this case, the integer is coerced to a Date instead of a double.

> c(Sys.Date(), 5)

However, this coercion is determined strictly by the type of the first argument.

> c(5, Sys.Date(), Sys.Date()+1)
[1] 5 16198 16199

Since type coercion is not commutative, it is generally inadvisable to rely on coercion rules to transform data. It is better to explicitly convert types as the intention of the operation becomes explicit.

### 2.1.5 Concatenation

What we've called the vector constructor is actually a special case of the concatenation function. Combining arbitrary vectors together is known as
concatenation. When all the elements of $c$ were scalars, it was convenient to call this vector construction. However, since scalars are simply vectors of length one, concatenation operates on arbitrary vectors and is the mechanism for adding elements to a vector. This function differs from languages that provide explicit semantics for arrays again thanks to vectorization. Since each argument to $c$ is a vector, new vectors can be created quickly without the need for looping.

**Definition 2.1.4.** Given vectors $x = \langle x_1, x_2, \ldots, x_m \rangle$, $y = \langle y_1, y_2, \ldots, y_n \rangle$ the concatenation of $x$ and $y$ is the ordered union of $x$ and $y$. It is denoted

$$c(x, y) = c(c(x_1, x_2, \ldots, x_m), c(y_1, y_2, \ldots, y_n))$$

$$= c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$$

$$= \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n \rangle$$

**Example 2.1.11.** Since all primitives are vectors, concatenation operates on any number of vectors with arbitrary length.

```r
> c(1:3, 5, 7, c(11, 13, 17, 19))
[1] 1 2 3 5 7 11 13 17 19
```

**Proposition 2.1.5.** The definition of concatenation implies the following statements $\forall$ vectors $x, y$ of the same type.

(a) Concatenation is idempotent: $c^{(m)}(x) = x \forall m \in \mathbb{N}$.

(b) Concatenation preserves ordering. It follows that concatenation is not commutative: $c(x, y) \neq c(y, x)$.

(c) Concatenation is associative: $c(x, c(y, z)) = c(c(x, y), z)$. Furthermore, $c(x, c(y, z)) = c(x, y, z)$.

(d) Length is linear: $|c(x, y, \cdots)| = |x| + |y| + \cdots$.

**Proof.** Let $x = \langle x_1, x_2, \ldots, x_m \rangle$, $y = \langle y_1, y_2, \ldots, y_n \rangle$, where $m, n \in \mathbb{N}$. Then

(a) For $m = 1$, $c(x) = x$. Assume that $c^{(m)}(x) = x$. Then

$$c^{(m+1)}(x) = c(c^{(m)}(x))$$

$$= c(x)$$

$$= x$$

(b) $c(x, y) = \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n \rangle$. On the other hand, $c(y, x) = \langle y_1, y_2, \ldots, y_n, x_1, x_2, \ldots, x_m \rangle$. Therefore $c(x, y) \neq c(y, x)$.
(c) Let $x, y$ defined as in (a). Let $z = \langle z_1, z_2, \ldots, z_p \rangle$. Then
\[
c(x, c(y, z)) = c(x, c(c(y_1, y_2, \ldots, y_n), c(z_1, z_2, \ldots, z_p))) \\
= c(x, c(y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p)) \\
= c(c(x_1, x_2, \ldots, x_m), c(y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p)) \\
= c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p) \\
= \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p \rangle
\]
And
\[
c(c(x, y), z) = c(c(x_1, x_2, \ldots, x_m, c(y_1, y_2, \ldots, y_n), z) \\
= c(c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n), z) \\
= c(c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p)) \\
= c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p) \\
= \langle x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n, z_1, z_2, \ldots, z_p \rangle = c(x, y, z)
\]
Therefore $c(x, c(y, z)) = c(c(x, y), z)$.

(d) Define vectors $x^1, x^2, \ldots, x^n$. By definition, $|c(x)| = |x|$. Assume that $|c(x^1, x^2, \ldots, x^n)| = |x^1| + |x^2| + \cdots + |x^n|$. Now let $a = c(x^1, x^2, \ldots, x^n)$ and $b = c(a, x^{n+1})$. Then $|b| = |c(a, x^{n+1})| = |a| + |x^{n+1}|$.

\[\square\]

Example 2.1.12. Idempotency is an important concept in both mathematics and computer science. Situations may arise where a sequence of concatenations is performed. Knowing the result in advance is useful but also knowing how a result appeared is often critical. For example, $c(c(c(x))) = x$, which is thematically similar to $(x^1)^1 = x$. Rewriting exponentiation as a function, the general form is exposed. Let $f(x, y) = x^y$. Then $f(f(x, 1), 1) = x$. The function $f$ is idempotent when $y = 1$. In a similar vein, concatenation on a single argument is idempotent. Therefore, if $c(x, y) = x$, we know that $y = 0$.

Example 2.1.13. Commutativity is useful for algebraic operations, but in the case of concatenation it would render it useless. Preserving order is an essential part of the process and provides a reliable way to access vector elements based on their ordinality. The diabetes dataset contains a number of features. We want to conduct a time-series analysis on feature 33 (regular insulin dose), which requires the data to be ordered based on date and time. Assume that $x$ contains data related to feature 33 and is already ordered. As new data is collected, it is stored in a new variable $y$. To perform the analysis on the union of $x$ and $y$, we want $c(x, y)$, which is clearly different from $c(y, x)$.

Example 2.1.14. We can see that associativity is a natural extension of the preservation of ordering. Hence $c(c(c(1:3, 5), 7), 11, 13, 17, 19) = c(1:3, c(5, c(7, c(11, 13, 17, 19))))$. 

\[\square\]
Example 2.1.15. The linearity of length extends to concatenation of an arbitrary number of arguments. Based on example 2.1.14, we can see how this works in practice. Given \( c(1 : 3, c(5, c(7, c(11, 13, 17, 19)))) \), the length of the resulting vector is computed as

\[
|c(1 : 3, c(5, c(7, c(11, 13, 17, 19))))| = |c(1 : 3, 5, 7, c(11, 13, 17, 19))|
\]

\[
= |1 : 3| + |5| + |7| + |c(11, 13, 17, 19)|
\]

\[
= 3 + 1 + 1 + 4
\]

\[
= 9.
\]

We made use of property 2.1.5c to flatten the nested vector structure, which makes it easier to compute the length. We could have just as easily computed the length moving left-to-right in the evaluation.

\[
|c(1 : 3, c(5, c(7, c(11, 13, 17, 19))))| = |1 : 3| + |5| + |c(7, c(11, 13, 17, 19))|
\]

\[
= 3 + 1 + 1 + 4
\]

\[
= 9.
\]

When vectors are concatenated only the ordering is preserved in the resultant vector but not the actual ordinal positions (excepting the first argument). Since length is a linear operator we can deduce the ordinal positions of the resulting vector given the arguments to concatenation. The following proposition tells us how to map indices on pre-concatenated vectors into the corresponding indices of the concatenated result.

Definition 2.1.6. Let \( x \) be a vector where \( |x| = n \) and let \( i \in \mathbb{N}_n \). The ordinal position of \( x_i \) in \( x \) is \( \text{ord}(x_i) = i \).

Proposition 2.1.7. Let \( x^1, x^2, \ldots, x^n \) be vectors of arbitrary length, where \( |n| > 1 \). For \( y = c(x^1, x^2, \ldots, x^n) \), \( \text{ord}(y_i) = \sum_{j=1}^{k} |x^j| + \text{ord}(x^{k+1}) \), where \( k = \arg\max_k \sum_{j=1}^{k} |x^j| < |y| \) and \( m = |y| - \sum_{j=1}^{k} |x^j| \).

Proof. Pending \( \square \)

Example 2.1.16. Taking Example 2.1.11 as a cue, the corresponding R code provides an anecdotal confirmation of the proposition.

```R
> x <- c(1,2,3)
> y <- c(11, 13, 17)
> a <- c(x, 5, 7, y)
> a[length(x) + 1 + 1 + 2] == y[2]
[1] TRUE
```
load_diabetes <- function(bad=c(2,27,29,40), base='.') {
  load.one <- function(i) {
    path <- sprintf('%s/data-%02i',base,i)
    flog.info("Loading file %s", path)
    o <- read.delim(path, header=FALSE, colClasses=c('character','character','numeric','numeric'))
    colnames(o) <- c('date','time','feature','value')
    o$date <- as.Date(o$date, format='%m-%d-%Y')
    o
  }
  idx <- 1:70
  lapply(idx[-bad], load.one)
}

FIGURE 2.2: Loading the diabetes dataset

Example 2.1.17. When data are collected and aggregated, it is difficult to keep track of indices. At times certain records may have been identified as important and their ordinal positions recorded. Parallelization is such a case where it may be desirable to process a dataset in chunks. Other times, data is provided in chunks and must be joined at a later time. This is the case with the diabetes dataset [?], which is loaded via load_diabetes as defined in Figure 2.2.4 Let’s look at all the values associated with pre-breakfast glucose (feature 58). When conducting the analysis on a single feature, it might be nice to consolidate the values into a single vector.

\begin{verbatim}
> y <- lapply(diabetes, function(x) x$value[x$feature==58])
> z <- do.call(c, y)
\end{verbatim}

Suppose we want to know from which datasets the tails of the distribution come from. We can determine this by using Proposition 2.1.7 to associate the lengths between the vectors. First let’s extract the ordinal positions of some outliers.

\begin{verbatim}
> outliers <- which(z > 400)
\end{verbatim}

The file that contains each outlier is simply the first file where the cumulative length is greater than the ordinal of the outlier.

\begin{verbatim}
> lengths <- cumsum(sapply(y, length))
> sapply(outliers, function(o) which.max(lengths >= o))
\end{verbatim}

\begin{verbatim}
[1] 12 12 16 16 20 20 20 20 49 60 64
\end{verbatim}

This approach balances ease of use of an ad hoc analysis while simultaneously preserving information necessary to locate outliers in raw data files.

In Section 2.1.3 we saw how the type of a vector can change when a vector is

\footnote{Some of these files have bad data and consequently are omitted from the analysis.}
constructed with arguments of mixed types. This is an exceptional situation that results in an operation that is not closed with respect to some of the operands. However, the atomic types are closed under concatenation when all arguments of the same type. To see how this works, we need to think of vectors as sets.

**Proposition 2.1.8.** Atomic vectors are closed under concatenation up to $k$. Let $x, y$ be vectors where $\text{typeof}(x) = \text{typeof}(y) = T \in \mathbb{T}$. Then $\text{typeof}(c(x, y)) \in T$.

**Proof.** Consider $x$ and $y$ as sets containing elements of $T$. The power set $P(T)$ is the set of all possible sets of type $T$. Define $a = c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n) \in P(T)$. Since $c(x, y) = c(x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$, $c(x, y) = a$ and $c(x, y) \in P(T)$.

The significance of Proposition 2.1.8 is that it acts as a null hypothesis of sorts. When this property doesn’t hold, we know that the set of argument types contained more than one type.

**Example 2.1.18.** It turns out that the diabetes dataset contains some bad values. In each file, column 4 contains the observed value for each measurement. We expect this vector to be numeric, but in file `data-02` a factor is returned instead.

```r
> x <- read.delim('data-02', header=FALSE)
> class(x$value)
[1] "factor"
```

Since concatenation is closed when all elements are of the same type, we know that there must be bad data in the file. Indeed, some records contain a value of $0\text{Hi}$ or $0\text{Lo}$, so R coerced the column first to character and then to factor.

**Exercise 2.1.** Why did the last statement in Example 2.1.10 get coerced to integer and not double?

---

### 2.2 Set theory

Most data are readily interpreted as sets. Panel data in particular can easily be partitioned into disjoint sets. In the diabetes dataset each person can represent a panel. It is clear that the panels are disjoint and the union composes the original dataset. Another example is time series data with rolling windows that can be modeled as overlapping sets. Here the sets are not disjoint, so the windows do not partition the original set. Nonetheless, the original set is a subset of the union of rolling windows. We can see that using the language of

---

5The maximum length is due to finite machine limits.
sets to describe data is natural, and R provides a rich vocabulary to do just this.

2.2.1 The empty set

Befitting a set-theoretic discussion, let’s first consider the empty set.

**Definition 2.2.1.** A vector with 0 elements is called the *empty set* and is denoted as \( c() \equiv \emptyset = \texttt{NULL} \). As expected, the cardinality of the empty set is \(|c()| = 0\).

One difference between the R version of the empty set and the mathematical empty set, is that the empty set is not preserved in concatenation. Hence, the empty set acts as the identity for the concatenation function: \( c(c(), x) = c(x) = x \) and \( c(x, c()) = x \).

This means that \( \emptyset \) cannot be an element of a vector and cannot be tested for set membership.

\[
\begin{align*}
> & \texttt{c()} \in c(3, 4, c()) \\
& \texttt{logical(0)}
\end{align*}
\]

The empty set affects concatenation in other ways, too. As a consequence of being the identity, the empty set satisfies commutativity over concatenation. This can create a conundrum if one expects the cardinality of the input and output to be consistent. For example, suppose some function \( f \) operates on a vector \( x \). If one of the results is \( \emptyset \), the output will have a cardinality one less
than the input cardinality. Even though the operation preserves ordering, the ordinals are not preserved. This loss of information means that it is not clear how to construct the mapping between input and output vectors.

**Example 2.2.1.** Suppose a function $f$ operates on a vector, returning the below result. In the output, what is $\text{ord}(7)$? We cannot know, since the ordinals were not preserved.

```r
> f(c(1,2,3,5,8,13))
[1] 7 3 6 9 11
```

**Example 2.2.2.** Despite this wrinkle caused by $\emptyset$, note that the calculation of ordinal positions is computable if the original ordinal of $\emptyset$ is known. Starting with Example 2.2.1, define $b <- c(x, c(), y)$. The ordinal position of $y_2$ in $b$ is $|x| + 0 + \text{ord}(y_2) = 5$. So $b[5] == y[2]$.

### 2.2.2 Set membership

We generally think of sets as comprising a collection of elements. These elements are said to be in the set. Set membership is represented mathematically by the symbol $\in$, whereas in R the `%in%` operator is used. On the surface the semantics may seem the same but there are subtle distinctions between the two. For a given element $a$ and set $A$, the expression $a \in A$ acts as a statement of fact or condition to be satisfied. Compare this to a $\%\in\% A$, which is a logical test that produces a boolean value. These two forms converge when we use the expression $\forall a \in A$, which describes a set. This latter expression maps more closely to the semantics of the

**Example 2.2.3.** Let’s use the diabetes dataset to illustrate this more clearly. In this dataset, data were collected using two different techniques. While an automatic recording device captured events precisely, paper records only captured the timestamp of a measurement based on four "logical timel slots". To find the records associated with the paper records can be accomplished using set membership.

```r
> timeslots <- c('8:00','12:00','18:00','22:00')
> subset(diabetes[[1]], time %in% timeslots)
   date    time feature value
21 1991-04-24 12:00    33    4
53 1991-04-29 12:00    65    0
66 1991-04-30 22:00    65    0
71 1991-05-01 12:00    65    0
95 1991-05-04 12:00    33    5
... 
```

You may have noticed that from a language perspective there isn’t much of a difference between an element of a set and a set. This is clearly due to the fact that all atomic types are vectors. Like other operators in R, `%in%` is vectorized.
The implication is that multiple elements can be tested for membership in a single operation. Testing for set containment is simply aggregating over set membership of each element in question.

**Definition 2.2.2.** Given sets $A$ and $B$, $A \subseteq B$ iff $\forall a \in A, a \in B$.

**Example 2.2.4.** Let $x = c(5,6,7)$ and $A = 1:10$. The set $x$ is a subset of $A$ iff $x_i \in A \forall x_i \in x$.

```r
> all(x %in% A)
[1] TRUE
```

Since each operand to `%in%` is a vector, it may be tempting to reverse the operands. However, set membership is not commutative.

```r
> all(A %in% x)
[1] FALSE
```

### 2.2.3 Set comprehensions and logic operations

A powerful method for constructing sets is the set comprehension. The general syntax is $\{x \mid \Phi(x)\}$, which is read "the set of values $x$ such that $\Phi(x)$ is true". For example, the natural numbers can be extracted from the integers using the following set comprehension: $\{x \mid x \in \mathbb{Z} \land x > 0\}$. The notation in R is similar but is constructed from a specific domain, which indicates that the set membership clause is implied.

```r
z <- -1000:1000
z[z > 0]
```

Like set notation, the R syntax supports logical operations to further specify the set membership rules. The standard set of boolean operators are valid in addition to the aggregation operators, as shown in Table 2.1. Since the goal is to construct a set from another set, the vectorized version of the boolean operators must be used. Constructing set comprehensions like this is useful for removing extreme values from data or selecting a specific slice of data. While the **all** and **any** operators are vectorized, notice that they return scalars instead of vectors. These functions are useful for detecting a specific condition within a set, such as an outlier or an unwanted value.

**Example 2.2.5.** Define $y <- x[x > 3]$. If $|y| > 0$, then **any**($x > 3$) is true.

**Example 2.2.6.** In one of the panels of the diabetes dataset, suppose we want to examine data for feature 69 (typical exercise activity) but only for times before lunch. This requires two logical conditions to be met, which is described as $\{x \mid x$ feature $= 69 \land x$ time $< 12 : 00\}$. The corresponding R expression is $x[x$ feature $== 69 \land \text{hour}(x$ time $) < 12,$].

Note that an expression that returns the empty set yields a vector of length 0. This behavior implies that in R the empty set is not unique! While a length
Table 2.1: Logical operators given input(s) of length n

<table>
<thead>
<tr>
<th>Operator</th>
<th>Logic</th>
<th>R</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>(\neg)</td>
<td>!</td>
<td>(n)</td>
</tr>
<tr>
<td>conjunction</td>
<td>(\land)</td>
<td>&amp;</td>
<td>(n)</td>
</tr>
<tr>
<td>disjunction</td>
<td>(\lor)</td>
<td></td>
<td>(n)</td>
</tr>
<tr>
<td>all</td>
<td>(\bigwedge_k)</td>
<td>all()</td>
<td>1</td>
</tr>
<tr>
<td>any</td>
<td>(\bigvee_k)</td>
<td>any()</td>
<td>1</td>
</tr>
</tbody>
</table>

0 vector is conceptually equivalent to c(), they are technically different. The reason is that c() has no type information, while a 0-length vector preserves type.

Example 2.2.7. Suppose we want to extract all samples associated with an non-existent feature. This results in a 0-length integer vector.

\[
\text{\texttt{diabetes$value[diabetes$feature==37]}}
\]

\[
\text{integer(0)}
\]

2.2.4 Set complements

In Example 2.2.3 we identified the records associated with the paper records. Suppose we want to find the complement, which are all the records associated with the automatic recording device. Suppose we want to only use the panels without these outliers. What we want is the complement, or negation, of the original set of this set, which is \(\forall a \notin A\). The ! (bang) operator is used for this purpose when using the \%in\% operator. The mechanical operation is unary logical negation, and when applied to \%in\% becomes set negation. It is now possible to show that the theorem \(A \bigcup \bar{A} = X\), where \(A \subseteq X\) holds with the R operators.

To simplify the analysis we’ll only look at the time component of one of the panels, so \(x \leftarrow \text{diabetes[[1]]}$time\). Now we can construct the set \(a\), which contains the times associated with the paper records and \(ac\) as its complement.

\[
a \leftarrow x[x \%in\% \text{timeslots}]
\]

\[
ac \leftarrow x[! x \%in\% \text{timeslots}]
\]

The union is simply the concatenation of these two subsets.

\[
y \leftarrow c(a, ac)
\]

The only task remaining is to verify that the two sets are equal. Unfortunately since vectors are ordered tuples, this ordering adds some complexity. Or does it? Set theory actually tells us how to test for set equality and the same approach is valid in R! Recall that two sets are equal if they are subsets of each other. Hence, \(X = Y\) iff \(X \subseteq Y\) and \(Y \subseteq X\). In R, we have

\[
\text{all}(x \%in\% y) \&\& \text{all}(y \%in\% x)
\]
Another form of set negation is based on a negative index. The negative index removes the given ordinal positions from a set, which can also be viewed as set subtraction. Consider a dataset \( x \) whose elements correspond to \( X = \mathbb{N} \). We can construct a subset \( Y \subset X \) such that \( \overline{Y} = X - Y \). Since \( Y \) can similarly be described by its index, notice the congruence between set subtraction and the set complement.

**Definition 2.2.3.** A negative index on a vector \( x \) is equivalent to set negation and is denoted \( x[\overline{i}] \equiv \{ x_k | k \notin i \} \) where index order is preserved. In other words \( x[\overline{i}] = c(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n) \). Due to the duality of scalars and vectors of unit length, this definition extends to any vectors in \( \mathbb{N}_k \), where \( k \leq |x| \). Given a vector \( y \subset \mathbb{N}_k \), \( x[\overline{y}] = x - \{ x_i \}_{i \in y} \).

This definition shows us that a vector of negative indices negates this subset within the context of the original set.

**Example 2.2.8.** In Figure 2.2 we remarked that the diabetes dataset contains panels with bad data. Instead of removing them in the `load_diabetes` function, suppose we loaded everything and now need to remove the panels prior to performing our analysis. To remove these rows from the dataset let’s define \( y \leftarrow c(2, 27, 29, 40) \). Then the filtered set is simply `diabetes[-y,]`.

**Proposition 2.2.4.** Let \( x \) be a vector of length \( n \) and \( k \in \mathbb{N}_n \). Then \( c(x[k], x[\overline{k}]) \neq x \).

**Proof.** This follows from the definition.

\[
c(x[k], x[\overline{k}]) = c(x_k, c(x_1, x_2, ..., x_{k-1}), c(x_{k+1}, ..., x_n))
\]
\[
= c(x_k, x_1, x_2, ..., x_{k-1}, x_{k+1}, ..., x_n)
\]

But \( x = c(x_1, x_2, ..., x_n) \), so \( c(x[k], x[\overline{k}]) \neq x \) \( \square \)

This proposition underscores the fact that the use of negative indices is not invertible. In set theory a set can be reconstructed after removal of elements: \( (A - B) + B = A \), which is possible because sets are unordered. With ordered sets, reconstructing the original set is cumbersome and requires splitting the dataset according to the original ordinal positions.

Negative indices preserve ordering in the sense that the resultant set is the same irrespective of the order of the negative indices.

\[
y_1 \leftarrow \text{sample}(y, 4)
\]
\[
\text{all}(x[-y] == x[-y1])
\]

**Example 2.2.9.** In Example 2.2.8 we removed the bad panels from the diabetes dataset. Suppose for a moment that a vector \( x \) contains the panel indices for the complete dataset. In other words, \( x \leftarrow 1:70 \). Another vector tracks the panels with bad data: \( \text{bad} \leftarrow c(2, 27, 29, 40) \). Let’s create a new vector that has the bad indices removed, such as \( y \leftarrow x[\overline{c(2, 27, 29, 40)}] \). How can we recreate the original vector at some arbitrary point in the future? Reconstructing
the original ordinals requires shifting \( y \) at the four given index positions. One way of doing this involves creating a 2D structure where each row represents the segments of the vector.

```r
segments <- cbind(c(1, bad), c(bad-1, length(x)))
x.new <- lapply(1:length(bad),
             function(idx) c(x[segments[i,1]], x[segments[i,2]], bad(idx)))
x.new <- do.call(c, x.new)
```

The complexity surrounding the reconstruction of the data is due to two factors. First is information loss. If the original ordinals were preserved, then the reconstruction would be far simpler. Suppose we recorded the original ordinals in the names attribute of \( x \), as

```r
names(x) <- 1:length(x)
```

Reconstructing the original vector is then a matter of sorting:

```r
x.new <- c(y, bad)
x.new[order(names(x.new))]
```

We’ll discuss this technique in more detail in Section 2.3.1. A more fundamental issue is that our attempt at reconstruction was based on a map process. For this situation what is needed is a fold process, which provides a means for iteratively reconstructing the vector instead of via segmentation. The distinctions between these two vectorization processes are discussed in Chapters 3 and 4.

### 2.3 Indexing and subsequences

The previous section treated vectors as sets and showed how to extract subsets from a set. In this section, we look at the mechanism of indexing and how it can be used to extract any subsequence from a vector. The indexing operator performs this function and is applied to an index vector [6] that represents the ordinal positions of the elements contained in the vector. As such the indexing operator supports many extraction methods determined by the type of the indexing operand. The most basic subsequence is created by specifying an index vector as a set of integers representing the ordinal positions of the target vector.

**Definition 2.3.1.** Let \( x \) be a vector, where \(|x| = n\). Let \( y \subseteq \mathbb{N}_n \), where \(|y| = m\). A subsequence of \( x \) is \( x[y] \equiv \langle x_{y1}, x_{y2}, \ldots, x_{ym} \rangle \). Given \( x = \langle x_k \rangle_{k \in \mathbb{N}} \), then

(a) a subsequence from the range \( i : j \) is \( x[i : j] = \langle x_k \rangle_{k=i}^{j} \);

(b) if \( i = j \) then \( \langle x_k \rangle_{k=i}^{i} = \langle x_i \rangle = x_i \).
For notational convenience, I use a shorthand notation to represent subsequences where \( \langle x_k \rangle_{k=i}^{j} \equiv x_{i:j} \). In this notation the subscript \( k \) is assumed and may be referenced in certain expansions.

**Example 2.3.1.** Let \( x \) be a vector \( c(1,1,2,3,5,8,13,21) \) and \( y = c(2,3,6) \). Then \( x[y] = c(1,2,8) \).

### 2.3.1 Named indices

We saw in Example 2.2.9 that the `names` attribute can be used to preserve the original ordinals of a vector. In addition to ordinals, indexing also supports character names. These names provide a mapping between unique characters and ordinal positions. Under the hood, named indices are implemented as a character vector of unique values that has the same cardinality as the target vector.

**Definition 2.3.2.** Let \( x \) be a vector. A character vector \( a \) is a named index of \( x \) if \( |a| = |x| \) and \( a_i \neq a_j \forall i,j \in \mathbb{N}_{|a|} \), where \( i \neq j \). Ordering is preserved between a vector and its named index, so elements of \( x \) can be extracted using \( a \) such that \( x[a_i] = x_i \).

A named index is associated with a vector using the `names` function. Using \( x \) as defined in 2.3.1, let’s associate the alphabet to this vector.

\[
> \text{names}(x) \leftarrow \text{letters}[1:|x|]
\]

We can now extract elements of \( x \) using the associated names.

\[
> x[c(\text{‘b’, ‘c’, ‘f’})]
> \text{b c f 1 2 8}
\]

The `names` function is peculiar insomuch that it is not vectorized and is also more liberal than the mathematical definition. For example, assigning only three names to the above vector yields a named index with compatible length where the remaining elements are NA.

\[
> \text{names}(x) \leftarrow \text{NULL}
> \text{names}(x) \leftarrow \text{letters}[1:3]
> x
> \begin{array}{cccccccc}
> a & b & c & <\text{NA}> & <\text{NA}> & <\text{NA}> & <\text{NA}> \\
> 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21
> \end{array}
\]

Another curiosity is that names do not technically need to be unique during assignment, but extracting elements by name will only return the first encountered element!

\[
> \text{names}(x) \leftarrow \text{c(rep(’a’,4), rep(’b’,4))}
> x[c(’a’, ’b’)]
> a b 1 5
\]
Example 2.3.2. In Example ?? we saw how it is difficult to reconstruct an ordered set after set subtraction. Using named indices, reconstruction becomes far simpler.

```r
> names(x) <- letters[1:length(x)]
> ns <- names(x)
> z <- c(x[-c(3,5,6)], x[c(3,5,6)])
> z
  a  b  d  g  h  c  e  f
1 1 3 13 21 2 5 8
> z[ns]
  a  b  c  d  e  f  g  h
1 1 2 3 5 8 13 21
```

This works because names are preserved over concatenation and subsetting operations. Hence the original ordinal structure can be retrieved from a saved copy of the original named index.

Since the `names` attribute is character-based, assigning the names with a vector of another type will automatically coerce them to character. The consequence is that comparison operators will be character-based, which may result in unexpected behavior. For instance, numeric indices will compare lexically instead of numerically.

```r
> x <- letters
> names(x) <- 1:length(x)
> x[names(x) < 13]
  "a" "j" "k" "l"
```

This doesn’t necessarily need to be an impediment so long as the lexical order is consistent with the ordering desired. A common trick to force lexical ordering of integers to match their natural ordering is to pad them with leading zeroes.

```r
> names(x) <- sprintf('%02i', 1:length(x))
> x[names(x) < 13]
  "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

Exploiting this property means that objects like `Dates` can be compared without issue. Otherwise, only the `%in%` operator would be available, which can complicate the set comprehension.

Named indices can be used to emulate a hash data structure. This is not a true hash, since lookup has $O(n)$ complexity. In essence, it is possible to create key-value pairs using a vector and a corresponding named index.

Example 2.3.3. In Example 2.2.3, we extracted all of the paper records using a vector that described the fictitious times. Suppose we want to find only those

---

6For $O(1)$ complexity, an `environment` object must be used.
records associated with lunch. Rather than using the time, we can create a key-value lookup so that the times can be accessed by word.

```r
timeslots <- c('8:00', '12:00', '18:00', '22:00')
names(timeslots) <- c('breakfast', 'lunch', 'dinner', 'bedtime')
subset(diabetes[[1]], time %in% timeslots['lunch'])
```

<table>
<thead>
<tr>
<th>date</th>
<th>time</th>
<th>feature</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 1991-04-24</td>
<td>12:00</td>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>53 1991-04-29</td>
<td>12:00</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>71 1991-05-01</td>
<td>12:00</td>
<td>65</td>
<td>0</td>
</tr>
<tr>
<td>95 1991-05-04</td>
<td>12:00</td>
<td>33</td>
<td>5</td>
</tr>
</tbody>
</table>

2.3.2 Logical indexing

Subsequences can also be created from logical vectors. Reasoning about this process requires the help of the inclusion function, which is inspired by the Kronecker delta.

**Definition 2.3.3.** Let \( a \) be a logical scalar and \( x \) a scalar. The *inclusion function* is defined:

\[
\delta_I(a, x) = \begin{cases} 
  x, & \text{if } a \text{ is true} \\
  \emptyset, & \text{if } a \text{ is false}
\end{cases}
\]

This function produces a value whenever the predicate \( a \) is true. In fact, the behavior of logical indices is predicated on order isomorphism insomuch that the logical vector must preserve the order of the original vector. Since these operations are vectorized, this assumption usually holds. However, when manipulating indices directly, this assumption may not necessarily hold.

The subsequence of a vector given a logical index vector can now be defined in terms of the inclusion function.

**Definition 2.3.4.** Let \( x \) be a vector where \( |x| = n \), and let \( a \) be a logical vector such that \( |a| = |x| \). Then \( x[a] = (\delta_I(a_k, x_k))_{k=1}^n \equiv \delta_I(a, x) \).

**Example 2.3.4.** When using a logical vector to extract a subset the length must be compatible with the target vector. Let’s use the Fibonacci sequence to illustrate the behavior of the inclusion function. Define \( x <- c(1,1,2,3,5,8,13,21,34) \). How do we extract the elements that are even? In a procedural language it is a matter of iterating over the vector, testing each value, and adding the matches to another vector.

```r
y <- c()
for (i in x) {
  if (i %% 2 == 0) y <- c(y, i)
}
```

The idiomatic approach is to construct a set comprehension with a logical test to extract only those elements that resolve to **true**.
Let's break it down into two pieces: the logical index and the subsetting operation. The logical index is simply a vector with the result of the modulus test for each element. Like other native operators, modulus is vectorized and returns a vector with the same length as its argument. The expression \( x \mod 2 == 0 \) yields the logical vector \( c(FALSE, FALSE, TRUE, \ldots) \). When the subsetting operation is applied to a logical vector with compatible length, its behavior is governed by the inclusion function. Hence \( \text{even} <- x \mod 2 == 0 \) implies \( x[\text{even}] = \delta_I(\text{even}, x) \).

**Proposition 2.3.5.** Let \( x \) be a vector where \( |x| = n \), and let \( a \) be a logical vector such that \( |a| = |x| \). Then \( 0 \leq |x[a]| \leq n \).

**Proof.** From the definition of \( \delta_I \), \( |\delta_I| = \begin{cases} 1, & \text{if } a \text{ is true} \\ 0, & \text{if } a \text{ is false} \end{cases} \)

The lower bound on \( |x[a]| \) occurs when \( a_k = false \forall k \in \mathbb{N}_n \). Then \( x[a] = \emptyset \) and \( |x[a]| = 0 \). Similarly the upper bound is reached when \( a_k = true \forall k \in \mathbb{N}_n \) and \( x[a] = x \) and \( |x[a]| = n \).

The fact that subsetting is a bounded operation implies that the ordinals of a subset are generally compatible with the original set. The significance of Proposition 2.3.5 is in helping to prove the following proposition.

**Proposition 2.3.6.** Let \( x \) be a sequence and \( a \) be a sequence of logical values, such that \( |a| = |x| \). Then \( \{x[a], x[-a]\} \) is a partition of \( x \).

**Proof.** Pending

**Example 2.3.5.** The \texttt{ifelse} function exploits this property to construct a conditional vector. At 0, the \texttt{sinc} function must be defined explicitly, since division by 0 is not allowed. This can be expressed by \texttt{ifelse(x == 0, 1, sin(x)/x}.

With this function, a vector is created for each alternative. The values selected are based on the properties of the partition, which are \( x[a] \cap x[-a] = \emptyset \) and \( x[a] \cup x[-a] = x \). Since these are sequences and not sets, note that the union preserves the ordering.

### 2.3.3 Ordinal mappings

With consistent ordering, we’ve seen how a set of ordinals can be used to associate elements between multiple sets. Set operations yield the elements satisfying the given expression, while the raw expression yields a logical vector. How then do you obtain the actual ordinal values associated with the logical vector? Enter the \texttt{which} function. The behavior of this function can be described using the inclusion function defined in Definition 2.3.2.

**Definition 2.3.7.** Let \( a \) be a logical vector with length \( n \). Then \( \text{which}(a) = \delta_I(a, \mathbb{N}_n) \).
Example 2.3.6. In the diabetes dataset, suppose we want to know which pre-breakfast glucose measurements are greater than $2\sigma$ from the mean.

\[
> \text{with(subset(diabetes[[1]], feature==58),} \\
> \quad + \text{which(abs(value - mean(value)) > 2 * sd(value))}
\]

\[
[1] 36 42 111
\]

It becomes clear that \texttt{which} is simply returning the ordinals associated with true values. Interestingly, this operation is context-free, meaning that ordinals constructed this way may not be appropriate for a given vector. As an extreme example, let’s extract the ordinals of a random logical vector via

\[
\text{which(sample(c(TRUE,FALSE), 20))}
\]

Now do these ordinals have any significance?

This line of questioning brings up another aspect of ordinal mappings, which is that a set of ordinals can be applied to multiple vectors if the vectors share the same context, such as with multivariate data. More generally, any two-dimensional dataset possesses this quality across columns (rows), meaning that a single set of ordinals applies consistently to any column (row).

Example 2.3.7. When extracting a specific feature from the diabetes dataset, this preservation of ordinals is used to drive the indexing. Let \( x \leftarrow \text{diabetes[[1]]} \). The values associated with feature 58 is obtained via

\[
x$value[x$feature == 58]
\]

Since each row contains a specific sample of data, the expectation is that the logical vector produced by the conditional expression has ordinals that map to the vector \( x$value \). Hence, to obtain the associated time measurements, the same logical vector can be used, such as \( x$time[x$feature == 58] \).

2.3.4 Sorting

In other languages sorting is done as a direct operation on a vector so that the result is the sorted vector. In R sorting relies on indexing, so the process is indirect. Basic sorting is performed by the \texttt{order} function. Instead of returning a modified version of the input vector \texttt{order} returns a vector of the corresponding ordinals. The second step is to extract a subset according to the new ordinal sequence. This two-step dance is a constant source of confusion for newcomers as it takes a non-traditional approach to a common computing problem. When viewed from the perspective of indexing, sorting becomes a natural extension to a fundamental semantic construct of the language.

Example 2.3.8. Suppose we want to order the observations for feature 58 according to time of day. We accomplish this by calling \texttt{order(x$time)}. The output is a set of ordinals where \( y_i \in N \mid x$time \\forall y_i \in y \). The second step is applying this ordering to the vector of actual values, which is \( x$value[y] \).

Now why might this scenic route be preferred over the more direct route? It
is debatable whether this extended syntax provides any visual pleasure so there must be another reason. Consider the process of data analysis. It is common to have numerous variables representing a set of features. In this situation all variables have the same length and have a consistent ordering such that the indices correspond to the same sample. By separating the ordinals from the actual mutation of a data structure means that the same ordered index can be applied to multiple vectors. Another advantage is that by unaltering the original vector, multiple orderings can be created quickly. Indeed, this is the basis for the `sample` function.

**Exercise 2.2.** Given \( x = (N.A, 1, 0, N.A, -1, -1) \), \( a = (TRUE, FALSE, FALSE, TRUE, FALSE, FALSE) \), reduce \( x[a] \) based on the inclusion function.

**Exercise 2.3.** Given \( x \) as defined in Exercise 2.2, \( P(x) = \begin{cases} \text{true}, & \text{if } x = 0 \\ \text{false}, & \text{otherwise} \end{cases} \), reduce the set comprehension \( \langle x | P(x) \rangle \) mathematically.

**Exercise 2.4.** Show \( x[a] \), where \( a \subset -N_k, |x| = k \).

## 2.4 Recycling

When two operands do not have the same length, recycling provides a mechanism for extending the shorter vector to the length of the longer vector. If the length of the shorter vector divides the longer vector, then the shorter vector is repeated until the lengths are equal. Recycling greatly enhances the power of vectorization. It takes a mathematical perspective to see why this concept is so powerful. Consider the addition of a scalar with a vector. The rule is to add the scalar value to each element of the vector, irrespective of the length of the vector. Hence, the scalar value is repeated until the length of the resulting vector is equal to the other operand. We typically don’t consider the mechanical aspect of this operation, although a close examination reveals its nature:

\[
a + \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} a + x_1 \\ a + x_2 \\ a + x_3 \end{bmatrix}
\]

Given a scalar \( a \) and vector \( x \), the notation is the same in R: \( a + x \). For a vector operand with arbitrary length \( n \), linear algebra thus gives rules for operands of length 1 and length \( n \), both of which divide \( n \). R simply takes this concept a step further and applies the recycling rule to any operand whose length divides the length of the longest vector.

The behavior of recycling can be codified with the `rep` function.

**Definition 2.4.1.** Let \( f : X^m \times Y^n \rightarrow Z^p \). If \( m \geq n \) and \( n \) divides \( m \),
Vector Mechanics

hour <- function(x) {
  parts <- strsplit(x, ':', fixed=TRUE)
  sapply(parts,
    function(p) as.numeric(p[1]) + as.numeric(p[2])/60)
}

FIGURE 2.4: Convert a time into a decimal hour

then \( y \rightarrow \text{rep}(y, \frac{n}{m}) \) and \( p = m \). Similarly if \( n > m \) and \( m \) divides \( n \), then \( x \rightarrow \text{rep}(x, \frac{n}{m}) \) and \( p = n \).

**Example 2.4.1.** Let \( x <- 3:6 \) and \( y <- 8:19 \). Then \( x + y = y + x \).

Vector lengths are not always compatible. If \( m \) does not divide \( n \), then the operation will not succeed and an error will be thrown:

```
Warning message:
In y + 1:5 :
  longer object length is not a multiple of shorter object length
```

Note that this also happens when a vector has 0 length, which can result from a bad calculation. In chapter ??, we’ll see how recycling affects vectorized operations.

Not limited to addition, recycling rules are honored for most arithmetic and algebraic operators by default. In addition to these operators, concatenation and assignment operators also support recycling. This is used most often within the context of two-dimensional data structures. For example, adding a scalar as a column to a matrix is done via

```
m <- matrix(rnorm(12), nrow=3)
cbind(m, 1)
```

A column can also be changed by assigning a scalar to it, which of course is recycled to the same dimension as a column in \( m \).

```
m[,2] <- 0
```

Recycling also applies to `data.frame`, which makes it easy to assign values to panel data. For example, suppose we want to add a column that specifies the closest meal associated with a measurement. We’ll compute this based on a decimalized hour as given by Figure 2.4.

```
lapply(diabetes, function(d) {
  d$meal <- NA
  d$meal[hour(d$time) < 11] <- "breakfast"
  d$meal[hour(d$time) >= 11 & d$time < 16] <- "lunch"
  d$meal[hour(d$time) >= 16] <- "dinner"
  d
})
```
Using this approach, data can be quickly added or updated in a manner that is clear and concise. This is the power of a declarative style.

**Example 2.4.2.** Suppose we want to find all outliers in the iris dataset. We construct a set comprehension to select only those records greater than six standard deviations. The first argument $abs$ is map-vectorized so $|abs(x)| = |x|$. On the other side of the comparison operator, $|6| = 1$ and $|sd(x)| = 1$. Hence $|6 \times sd(x)| = 1$. Now due to recycling, $|abs(x) > 6 \times sd(x)| = |x|$.

**Exercise 2.5.** Let $x = rnorm(10)$. Determine the length of the operation $(x - \bar{x})^2$.

### 2.5 Summary

This chapter explored the many facets of the vector type in R. Vectors are fundamental to the language and can be used to model numerous mathematical entities. We also introduced the concept of vectorization, which is developed further in subsequent chapters. Through vectorization, many mathematical operations translate directly to R, which simplifies the transition from mathematical ideas to computer code.