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 user. 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 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.

All functions are first-class in R. 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`.

```r
> 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
Sepal.Length  Sepal.Width  Petal.Length  Petal.Width
 5.843333 3.057333 3.758000 1.199333
```

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.\(^2\) The purpose of functions is to abstract

\(^2\)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

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.
library(randomForest)
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

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.

> 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.

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.

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
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

ksvm(Species ~ ., 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.

model <- setup_svm(kernel='besseldot')
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 NAs. 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)
model$classify(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-
cclusion of first-class functions in a language, the need for design patterns all
but disappears.
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

```r
t 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 complexity. Not surprisingly, functional programming provides a liberating alternative to the tyranny of all-encompassing class hierarchies. Rather than attempting 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. In Python, `with` operates on a callable object that has a `__enter__` and `__exit__` function defined.

```r
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. 4

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.
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.

> cat("line1
", file="example.data")
> using.resource <- setup_using(file("example.data"))
> using.resource(readLines)
[1] "line1"
> cat("line2
", 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 \( \langle \rangle \) 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)
  }
}

> 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.

\[
\begin{align*}
> & g \leftarrow \text{seq.gen}(5) \\
> & g() \\
& [1] 5 \\
> & g() \\
& [1] 6
\end{align*}
\]

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: \( \frac{df}{dx} \). 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 \( d(f) \equiv \frac{df}{dx} \) 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 \( f : X \times X \to X \) 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 \( \circ : X \times X \to X \). 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.

> `f + g (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 $v_1$, $v_2$, etc., the symbols $\lambda$ and $\cdot$, 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 $(MN) \in \Lambda$.

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 $\equiv$. 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 \( \lambda x.x \cdot 2 = \lambda y.y \cdot 2 \). 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 \( 5! = 5 \cdot 4! = 20 \cdot 3! = 60 \cdot 2! = 120 \). 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 <- \function(x) x + 1
> f(5) == (\function(x) f(x))(5)

[1] 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.

$$f = \lambda y. f(y)$$

$$= \text{function}(y) f(y)$$
Functions as a lingua franca

The lambda calculus also supports multivariate functions via Currying [2]. Additional arguments are appended after the λ symbol as $\lambda xy.X$.

**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

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

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.

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

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
Modeling Data With Functional Programming In R

TO_NAT <- function(x) x + 1

FIGURE 1.13: Mapping Church numerals to natural numbers

\[ F^0(M) = M \] and \[ F^{n+1}(M) = F(F^n(M)) \]. The Church numerals are then defined as \( c_n \equiv \lambda f.\lambda x. f^n(x) \). 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.

\[
\begin{align*}
C0 & \gets \text{function}(f) \text{ function}(x) \ x \\
C1 & \gets \text{function}(f) \text{ function}(x) \ f(x) \\
C3 & \gets \text{function}(f) \text{ function}(x) \ f(f(f(x)))
\end{align*}
\]

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.

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

When \( \text{SUCC} \) is applied to a Church numeral, we see the remarkable behavior that the Church numeral is acting as data and also a function simultaneously. Let’s apply the rules of the lambda calculus to see this more clearly. \(^5\)

\[
\text{SUCC} \ c_2 = \text{SUCC}[n := c_2] &= (\lambda n.\lambda f.\lambda x. f((n(f)(x))))[n := 2] \\
&= \lambda f.\lambda x. f((\lambda g.\lambda y. g(y))[g := f](x)) \\
&= \lambda f.\lambda x. f(\lambda y. f(y))[y := x] \\
&= \lambda f.\lambda x. f(f(f(x))) \\
&= c_3
\]

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.

\[
> \text{SUCC(C2)} \\
\text{function}(f) \text{ function}(x) \ f(n(f)(x)) \\
<\text{environment: 0x7fe521d88740}>
\]

\(^5\)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 \( PLUS = \lambda m \ n \ m \ SUCC \ n \). 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

\[
c_2 \ SUCC = (\lambda f \ x. f^2(x))[f := SUCC] \\
= (\lambda f \ x. f(f(x)))[f := SUCC] \\
= \lambda x. SUCC(SUCC(x)).
\]

Now apply c3 to this intermediate result, yielding

\[
(\lambda x. SUCC(SUCC(x))) \ c_3 = SUCC(SUCC(c_3)) \\
= c_5.
\]

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.