Portfolio 4

Sam Bowyer

Two important programming paradigms are those of *imperative* and *declarative* programming. Imperative programming relies on implementing algorithms through explicitly stating each of the steps required, changing the state of the program directly through these steps (much as imperatives in natural language form a command). In contrast, declarative programming languages control *how* a sequence of operations should be computed, leaving the programmer to write the logic of computation that expresses *what* needs to be accomplished (declarative programs can be seen as heavily related to formal logic deductions and/or sequences of mathematical functions).

`R` has the ability to work in both an imperative and declarative approach, we shall introduce some of these capabilities through object-oriented programming (an imperative paradigm) and functional programming (a declarative paradigm).

Object-Oriented Programming (OOP) In `R`

OOP is a very popular imperative paradigm that works by manipulating objects—these can be thought of as containers which have their own variables/data (called fields) and functions (called methods). Objects can come with any number of different fields and methods, specified by its *class*, which can be thought of as a blueprint for how to create a certain type of object (though the values stored inside these different objects need not be identical). Many objects can be created from the same class and in most OOP languages we can also create *child classes* which *inherit* the fields and methods of a *parent class*, with the ability to add extra fields and methods that the parent class will not have and fix the value of some fields from the parent class. This leads us to one of the main benefits of OOP: polymorphism. This describes the way in which code can be written to produce different results based on the type of object (the class) it is given as input (thus within this code the objects are referred to by a variable representing potentially *many forms/types*). With careful design, this hierarchical structure can greatly improve the ease of creating and maintaining large, sophisticated software systems.

S3

The first OOP model in `R` that we'll use is also the simplest and least truly object-oriented, it is largely built on conventions rather than on any rigorous checking of code which means that although it can be flexible it is harder to scale to larger, more sophisticated systems.

To create an object we'll first create a *constructor* function which sets up the relevant fields of an object (with values perhaps supplied by the user and checked by an external *validator* function) and then returns that object (as a list). The values of these fields can then be accessed and modified through *helper* functions ('getters' and 'setters'), though these don't always have to exist—in fact we won't define them here since for S3 the objects are really just lists, but we will revisit these in S4.

```
#' Constructor for `polynomial`.
#'
#' This function creates an object of the class `polynomial`
#'
#' @param coeffs - a vector of the polynomial's coefficients
#'                 (x^0's coefficient, then x^1's etc.)
#'
```

```r
#' @return an S3 object of type `polynomial`
polynomial <- function(coeffs) {
  # Use the validator function to check the given coefficients
  if (!checkCoeffs(coeffs)){
    stop("Invalid coefficients, should be a vector of numeric values only.")
  }

  # Define the order of the polynomial
  order = 0
  for (i in length(coeffs):1){
    if (coeffs[i] != 0){
      order = i-1
      break
    }
  }

  # This is the object we're creating
  p <- list(coeffs=coeffs, order=order)

  # This declares that the object 'p' is of the S3 class `polynomial`
  class(p) <- "polynomial"

  return(p)
}


#' Validator function for `polynomial` constructor.
#'
#' Checks that `coeffs` is a non-empty vector containing only numeric values
#'
#' @param coeffs - coefficients for the `polynomial` class to be checked
#'
#' @return
checkCoeffs <- function(coeffs){
  return(is.vector(coeffs) && all(sapply(coeffs, is.numeric)))
}

poly1 = polynomial(c(2,-1,1))
poly1$order
```

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

```r
poly1$coeffs
```

```
## [1]  2 -1  1
```

Now that we have defined the data in this way, S3 allows us to use generic functions such as `print` and `plot` on objects of type `polynomial` using the naming convention `[method_name].[class_name]()`.

```r
print.polynomial <- function(p){
   cat(p$coeffs[1])
  if (length(p$coeffs) > 1){
    for (i in 2:length(p$coeffs)){
      if (p$coeffs[i] >= 0) sign = " + " else sign = " "
       cat(sign, p$coeffs[i], "x^", i-1, sep = "")
```

```
    }
  }
}
print(poly1)
```

```
## 2 -1x^1 + 1x^2
```

We can also define a method of this class to evaluate the polynomial at a certain `x` value by first defining a generic version of it:

```
# The generic `evaluate` function
evaluate <- function(p, x) {
  UseMethod("evaluate")
}

# The `evaluate` method for the `polynomial` class
evaluate.polynomial <- function(p, x){
  total = rep(0,length(x))
  for (i in 1:length(p$coeffs)){
    total = total + p$coeffs[i] * x^(i-1)
  }
  return(total)
}
evaluate(poly1, 4)
```

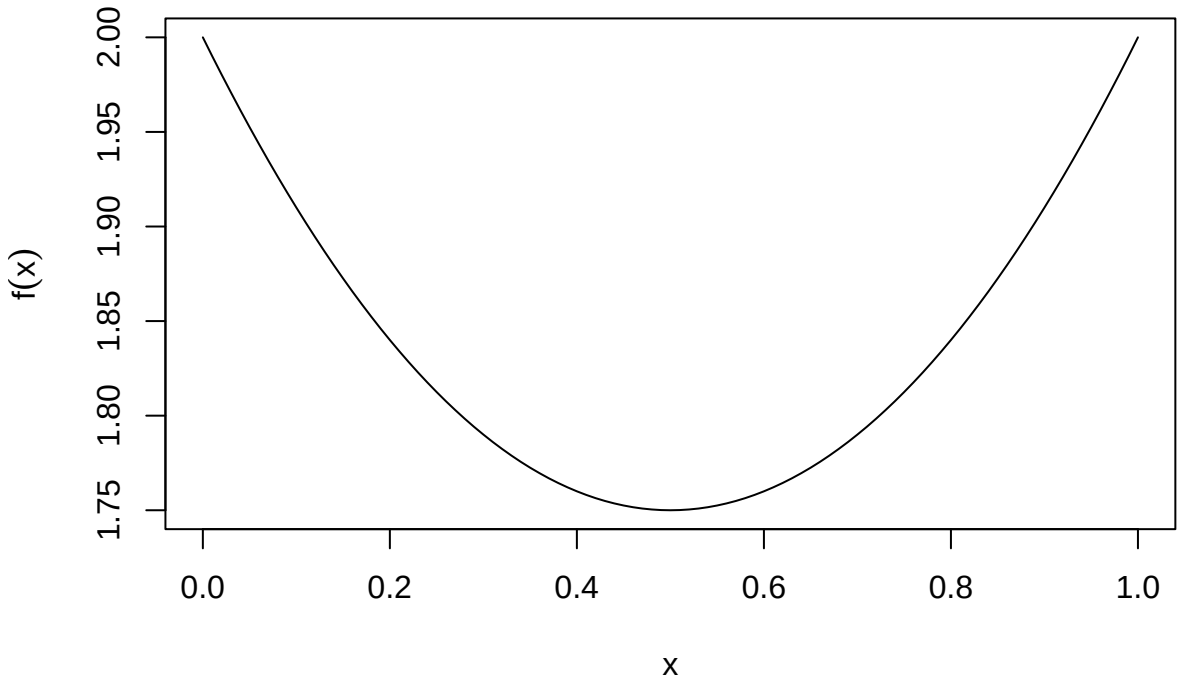
```
## [1] 14
```

```
evaluate(poly1, c(4,5))
```

```
## [1] 14 22
```

This then allows us to create a `plot` method for the `polynomial` class (note that `plot` is already a generic function in `R` so we don't need to define it in the way that we did for `evaluate`).

```
plot.polynomial <- function(p, y=NULL,...){
  plot(function(x) evaluate(p, x), xlab = expression(x), ylab = expression(f(x)))
}
plot(poly1)
```



S3 supports inheritance through the assignment of multiple classes to an object. Let us define the generic function `describe` for both the `polynomial` class and also a child class of `polynomial` named `cubic`.

```
# The generic `describe` function
describe <- function(p, x) {
  UseMethod("describe")
}

# The `describe` method for the `polynomial` class
describe.polynomial <- function(p, x){
  return(paste("This is a polynomial of order ", p$order, ".", sep=""))
}

# Define the child class `cubic`
cubic <- function(coeffs){
  p = polynomial(coeffs)

  if (p$order != 3){
    stop("Coeffs do not form a cubic.")
  }

  # This says that `p` is an object of class `cubic` which is a child
  # of the class `polynomial`
  class(p) <- c("cubic", "polynomial")
  return(p)
}
```

```
# The `describe` method for the `cubic` class
describe.cubic <- function(p, x){
  return(paste("This is a cubic."))
}
```

If we create a `cubic` object `poly2`, notice how it has a different class to `poly1`:

```
poly2 = cubic(c(3,-2,1,0.5))
class(poly2)
```

```
## [1] "cubic"      "polynomial"
```

```
class(poly1)
```

```
## [1] "polynomial"
```

Meaning when we call `describe` on `poly2` it refers to the `cubic` method rather than the `polynomial` class.

```
describe(poly1)
```

```
## [1] "This is a polynomial of order 2."
```

```
describe(poly2)
```

```
## [1] "This is a cubic."
```

Note, that `poly2` still has access the method `evaluate`, `print` and `plot` as we defined earlier—the child class `cubic` has inherited these methods from its parent and `R` goes up the class hierarchy until it finds a class of `poly2` for which these methods are defined.

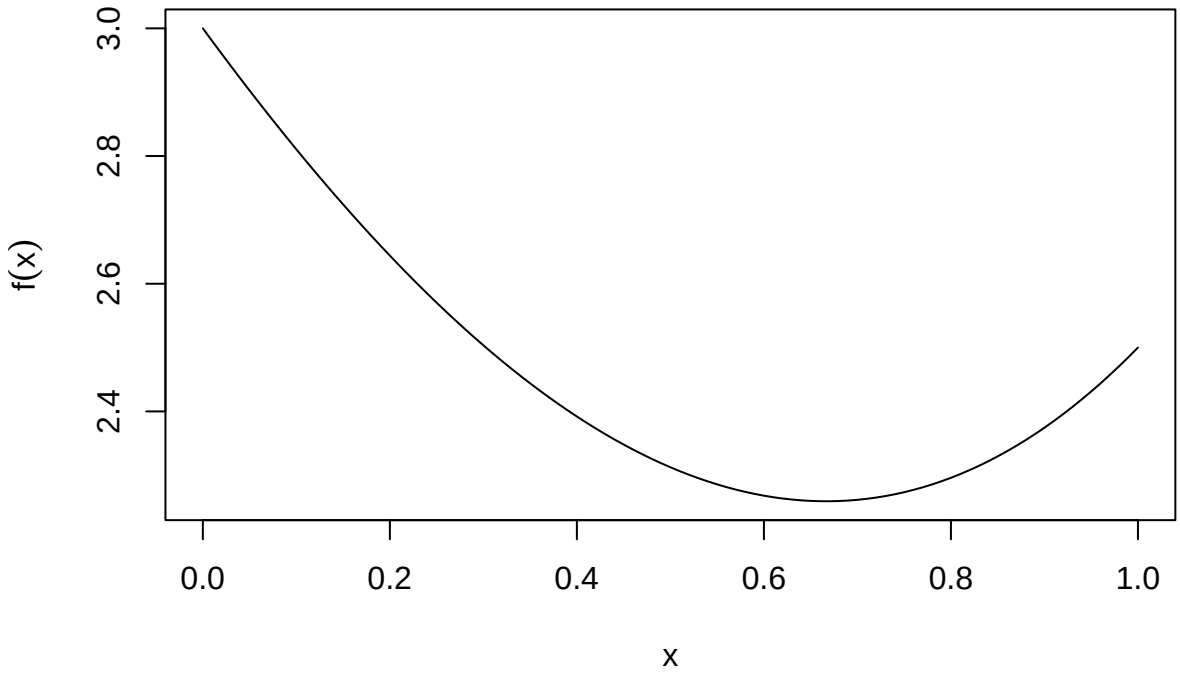
```
print(poly2)
```

```
## 3 -2x^1 + 1x^2 + 0.5x^3
```

```
evaluate(poly2, 4)
```

```
## [1] 43
```

```
plot(poly2)
```



S4

S4 is a more formal OOP model within `R` which requires a bit more structure in the code but as a result gives you a more robust system to work with. We can implement the same `polynomial` system as we did previously within S4 syntax. Note that we define classes and methods more explicitly with the `setClass` and `setMethod` functions, whilst validation can now be done with a specific `setValidity` function.

```
# This package is usually loaded by default but it is good to write it out
# to help someone reading the code later
library(methods)

# First define the fields (called `slots` in S4) of the class `polynomial`
setClass(Class="polynomial",
         slots = c(
           coeffs = "numeric",  # Slots must be defined by name and type
           order  = "numeric"
         )
)

# The constructor of a class must be called "initialize" and is called when
# an object is created with the function `new(...)` (here: `new(coeffs)`)
setMethod("initialize", "polynomial",
  function(.Object, coeffs) {
    # Define the order of the polynomial
    order = 0
    for (i in length(coeffs):1){
```

```
      if (coeffs[i] != 0){
        order = i-1
        break
      }
    }

    # Assign the values of the object slots using `.Object@`
    .Object@coeffs = coeffs
    .Object@order  = order

    return(.Object)
  }
)

setValidity("polynomial",
  function(p){
    if(is.vector(p@coeffs) && all(sapply(p@coeffs, is.numeric))) TRUE
    else stop("Invalid coefficients, should be a vector of numeric values only.")
  }
)
```

We can then define `print`, `evaluate` and `plot` methods for this class as follows:

```
setMethod("print", "polynomial",
  function(x) {
    cat(x@coeffs[1])
    if (length(x@coeffs) > 1){
      for (i in 2:length(x@coeffs)){
        if (x@coeffs[i] >= 0) sign = " + " else sign = " "
         cat(sign, x@coeffs[i], "x^", i-1, sep = "")
      }
    }
  }
)

# Since `evaluate` isn't already a generic function we have to first make it one
setGeneric("evaluate")
```

```
## [1] "evaluate"
```

```
setMethod("evaluate", "polynomial",
  function(p, x) {
    total = rep(0,length(0))
    for (i in 1:length(p@coeffs)){
      total = total + p@coeffs[i] * x^(i-1)
    }
    return(total)
  }
)

setMethod("plot", "polynomial",
  function(x) {
    plot(function(y) evaluate(x, y), xlab = expression(x), ylab = expression(f(x)))
  }
)
```

Let us now create a polynomial and check that these methods work.

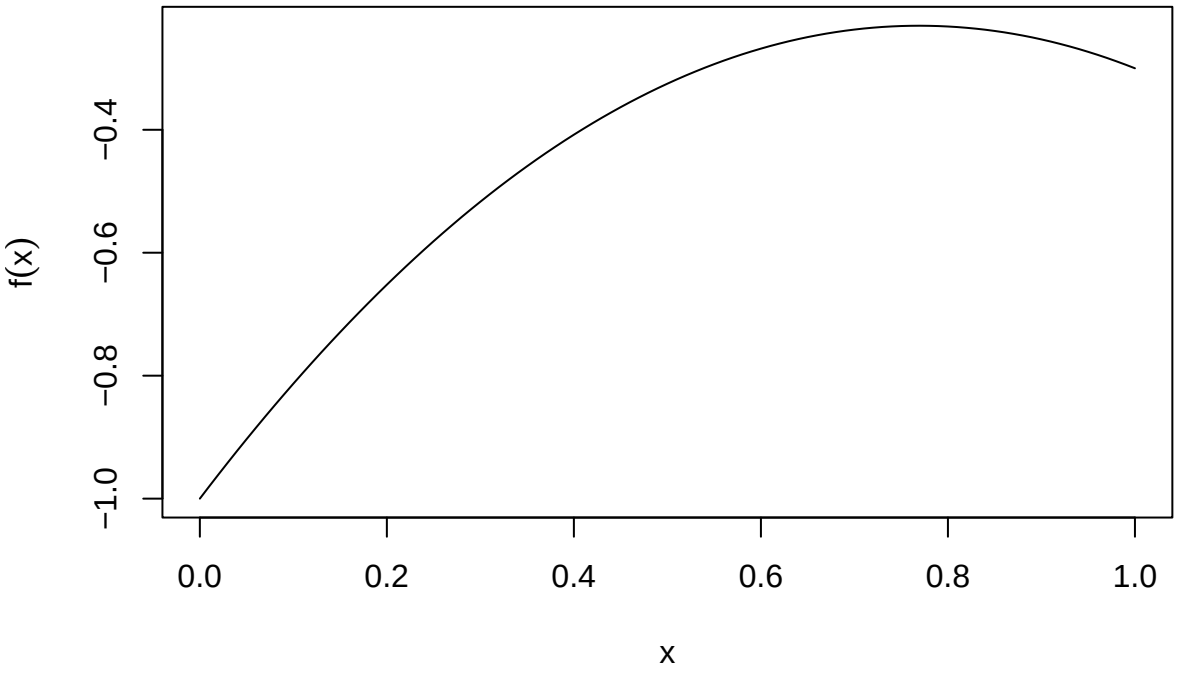
```
poly3 = new("polynomial", coeffs=c(-1,2,-1.3))
print(poly3)
```

```
## -1 + 2x^1 -1.3x^2
```

```
evaluate(poly3, c(1,2,3))
```

```
## [1] -0.3 -2.2 -6.7
```

```
plot(poly3)
```



With our S4 implementation of the `polynomial` polynomial class we will also include getters and setters which help to control what aspects of an object users have access to—fields/slots can have getters and/or setters or neither. As an example we'll add both a getter and a setter for `coeffs`:

```
setGeneric("coeffs", function(x) standardGeneric("coeffs"))
```

```
## [1] "coeffs"
```

```
setGeneric("coeffs<-", function(x, value) standardGeneric("coeffs<-"))
```

```
## [1] "coeffs<-"
```

```
setMethod("coeffs", "polynomial", function(x) x@coeffs)
setMethod("coeffs<-", "polynomial",
  function(x, value){
    x <- initialize(x, coeffs=value)
    validObject(x)
```

```
    return(x)
  }
)

print(coeffs(poly3))
```

```
## [1] -1.0  2.0 -1.3
```

```
# Now we change the linear coefficient
coeffs(poly3)[2] <- 0.5
print(coeffs(poly3))
```

```
## [1] -1.0  0.5 -1.3
```

S4 also allows for inheritance using the argument `contains` within `setClass` and even allows for classes to inherit from multiple parents (in which case the child class has all slots from all of its parents). The inheritance of methods works similarly as it does in S3 when inheriting from only one parent, however, with multiple parents it can get significantly more complicated (see Section 15.5 in Advances R for more details).

Reference Classes

Reference classes are the final OOP model we'll consider, wherein methods are defined as part of the class rather than as a class-specific instance of a generic function. This, along with modify-in-place semantics rather than S3/4's copy-in-place semantics, help improve the degree of encapsulation (the extent to which control of an object's data is separated from the rest of the program) of our OOP system.

The equivalent implementation of our `polynomial` class with reference classes would be:

```
# Again, this isusually loaded by default but useful for
# someone reading the code to see this explicitly
library(methods)

polynomialRC <- setRefClass("polynomialRC",
                            fields = c(coeffs="numeric", order="numeric"))

polynomialRC$methods(
  initialize = function(coeffs){
    # Define the order of the polynomial
    order = 0
    for (i in length(coeffs):1){
      if (coeffs[i] != 0){
        order = i-1
        break
      }
    }

    # Assign the values of the object slots using `.self$`
    .self$coeffs = coeffs
    .self$order  = order
  },

  print = function() {
    cat(.self$coeffs[1])
    if (length(.self$coeffs) > 1){
      for (i in 2:length(.self$coeffs)){
        if (.self$coeffs[i] >= 0) sign = " + " else sign = " "
```

```
        cat(sign, .self$coeffs[i], "x^", i-1, sep = "")
      }
    }
  },

  evaluate = function(x) {
    total = rep(0,length(x))
    for (i in 1:length(.self$coeffs)){
      total = total + .self$coeffs[i] * x^(i-1)
    }
    return(total)
  },

  plot = function() {
      curve(.self$evaluate(x) , xlab = expression(x), ylab = expression(f(x)))
  }
)
```

Note that we reference the object's fields with `.self$[field]` rather than S3's `$` and S4's `@` symbols, this is much more like C++ and Python. Defining all of the methods in a class in one place generally improves readability and is again more similar to class/method implementations in other OOP languages.

We can now create a `polynomial` object using the `polynomialRC$new` command and use its methods as follows:

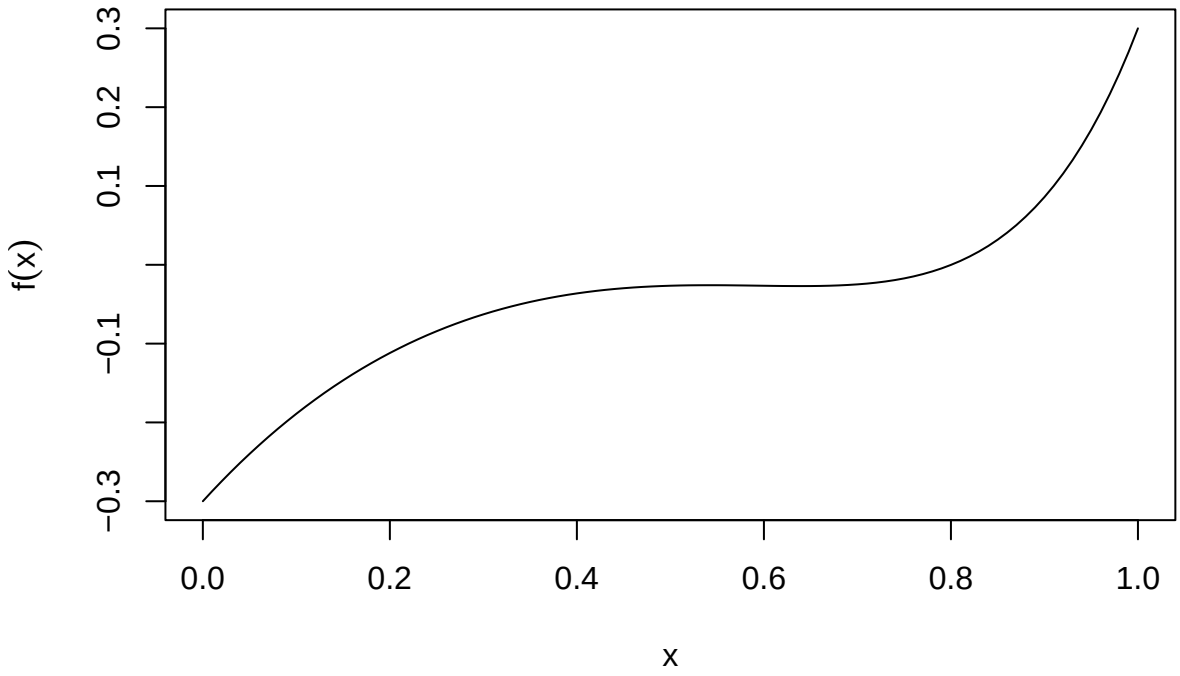
```
poly4 = polynomialRC$new(coeffs=c(-0.3,1.3,-2,1,0.1,-0.2,-0.6,1))
poly4$print()
```

```
## -0.3 + 1.3x^1 -2x^2 + 1x^3 + 0.1x^4 -0.2x^5 -0.6x^6 + 1x^7
```

```
poly4$evaluate(c(4,5,6))
```

```
## [1]  13784.1  68268.7 250668.3
```

```
poly4$plot()
```



Reference classes also allow for inheritance from multiple parent classes using the `contains` argument of the `setRefClass` function. Further discussion of this can be found in Advanced R here.

Functional Programming

Functional programming is a declarative programming paradigm in which programs are made from sequences of functions being applied or composed together. An important aspect of functional programming is the use of *pure functions*—functions which have no side effects, i.e. that don't change the global program state e.g. by printing output or plotting a graph. We typically aim to use only pure functions but sometimes side effects are needed to perform certain tasks, particularly in data analysis (e.g. reading or writing data, or generating pseudo-random numbers). `R` has many functional capabilities, some of which we've already used in the OOP section of this portfolio and previous ones (we gave `plot` the function `evaluate` as an argument and used `Map` and `apply` similarly). Importantly, `R` has *first-class functions*, meaning that functions can be stored in data structures, can be returned by other functions and can be arguments to other functions, meaning we can write code like this:

```
double <- function(f){
  function(...){
    return (2*f(...))
  }
}
f1 <- function(x) x^2+1
f2 <- function(x) x^2+2

funcs = c(f1, f2)
input = 1
```

```
for (f in funcs){
  print(f(1))
  print(double(f)(1))
}
```

```
## [1] 2
## [1] 4
## [1] 3
## [1] 6
```

Another important feature of `R` is that if we use a function `foo` to create another function `bar` the variables/arguments used in `foo` are no longer available outside of `foo`—that is, `foo` is a *closure*. For example, note that once we have created `bar`, `x` is no longer available to us.

```
foo <- function(x){
  bar <- function(y) x*y
}
b = foo(4)
b(5)
```

```
## [1] 20
```

```
exists('x')
```

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

This is an important feature in keeping code robust and making it easier to understand what individual variable names actually refer to (though this can still get confusing).

The final functional feature we will consider is *lazy evaluation*, in which `R` tries to evaluate only the parts of code which are necessary. Often this is a very useful feature and can speed up programs by reducing the number of times certain expressions are computed, however, it can lead to unusual results, particularly when we are dealing with functions that return functions. We can see an example of this using our functions `foo` and `bar` as follows:

```
x = 3
bar = foo(x)
x = 4
bar(4)
```

```
## [1] 16
```

You can see that our function `foo` doesn't actually consider `x`'s value when it creates the function `bar`, it only checks `x`'s value when we call `bar(4)`, by which time `x` has changed from `3` to `4`. To fix this we use the `force` command, which stops the lazy evaluation of its argument, resulting in a `bar` which uses the value of `x` that it receives when it is created.

```
foo <- function(x){
  force(x)
  bar <- function(y) x*y
}
x = 3
bar = foo(x)
x = 4
bar(4)
```

```
## [1] 12
```