CourseNotes_AdvR_Week4

Frank Fichtenmueller

6 November 2016

Object Oriented Programming in R

*Object oriented programming** is one of the most successful and widespread philosophies of programming and is a cornerstone of many programming languages including Java, Ruby, Python, and C++.

`R` has **three object oriented systems** because the roots of R date back to 1976, when the idea of object oriented programming was barely four years old. New object oriented paradigms were added to R as they were invented, so some of the ideas in R about object oriented programming have gone stale in the years since.

It's still important to understand these older systems since a **huge amount of R code is written with them**, and they're still useful and interesting! Long time object oriented programmers reading this book may find these old ideas refreshing.

The **two older object oriented systems** in R are called - `S3` and - `S4`, and the

*modern system** is called

- `RC` which stands for "reference classes." Programmers who are already familiar with object oriented programming will feel at home using RC.

General Principles of OOP

There a several key principles in object oriented programming which span across R's object systems and other programming languages.

Class | Objects | Attributes | Functions

The first are the ideas of a **class and an object**. The world is made up of physical objects - the chair you're sitting in, the clock next to your bed, the bus you ride every day, etc.

Just like the world is full of physical objects, your programs can be made of objects as well. A class is a blueprint for an object: it describes the parts of an object, how to make an object, and what the object is able to do. If you were to think about a class for a bus (as in the public buses that roam the roads) this class would describe attributes for the bus like the number of seats on the bus, the number of windows, the top speed of the bus, and the maximum distance the bus can drive on one tank of gas.

Buses in general can perform the same actions, and these actions are also described in the `class`: a bus can open and close its doors, the bus can steer, and the accelerator or the brake can be used to slow down or speed up the bus. Each of these actions can be described as a method which is a function that is associated with a particular class. We'll be using this class in order to create individual bus objects, so we should provide a constructor which is a method where we can specify attributes of the bus as arguments. This constructor method will then return an individual bus object with the attributes that we specified.

Inheritance

You could also imagine that after making the bus class you **might want to make a special kind of class for a party bus**.

Party buses **have all of the same attributes and methods as our bus class**, but they *also have additional attributes and methods* like the number of refrigerators, window blinds that can be opened and closed, and smoke machines that can be turned on and off. Instead of rewriting the entire bus class and then adding new attributes and methods, it is possible for the party bus class to inherit all of the attributes and methods from the bus class. In this framework of inheritance, we talk about the bus class as the super-class of the party bus, and the party bus is the sub-class of the bus. What this relationship means is that the party bus has all of the same attributes and methods as the bus class plus additional attributes and methods.

Summary

- R has three object oriented systems: `S3`, `S4`, and `Reference Classes`.
- `Reference Classes` are the **most similar to classes and objects in other programming languages**.
- `Classes` are blueprints for an object.
- `Objects` are individual instances of a class.
- `Methods` are functions that are associated with a particular class.
- `Constructors` are methods that create objects.
- **Everything in R** is an `object`.
- `S3` is a liberal object oriented system that allows you to assign a class to any object.
- `S4` is a more strict object oriented system that build upon ideas in S3.
- `Reference Classes` are a modern object oriented system that is similar to Java, C++, Python, or Ruby.

S3

Conveniently everything in R is an object. By "everything" I mean every single "thing" in R including numbers, functions, strings, data frames, lists, etc. If you want to know the class of an object in R you can simply use the class() function:

```
class(2)
```

```
## [1] "numeric"
```

```
class("is in session.")
```

```
## [1] "character"
```

```
class(class)
```

```
## [1] "function"
```

Now it's time to wade into some of the quirks of R's object oriented systems. In the S3 system you can arbitrarily assign a class to any object, which goes against most of what we discussed in the Object Oriented Principles section.

Class assignments can be made: - using the `structure()` function, or you can - assign the class using `class()` and `<-` :

```
special_num_1 <- structure(1, class = "special_number")
class(special_num_1)
```

```
## [1] "special_number"
```

```
special_num_2 <- 2
class(special_num_2)
```

```
## [1] "numeric"
```

```
class(special_num_2) <- "special_number"
class(special_num_2)
```

```
## [1] "special_number"
```

Creating a constructor Function in R

This is completely legal R code, but if you want to have a better behaved S3 class you should create a **constructor which returns an S3 object**.

The `shape_S3()` function below is a constructor that returns a `shape_S3 object`:

```
shape_s3 <- function(side_lengths){
  structure(list(side_lengths = side_lengths), class = "shape_S3")
}
square_4 <- shape_s3(c(4, 4, 4, 4))
class(square_4)
```

```
## [1] "shape_S3"
```

```
triangle_3 <- shape_s3(c(3, 3, 3))
class(triangle_3)
```

```
## [1] "shape_S3"
```

We've now made two shape_S3 objects: square_4 and triangle_3, which are both instantiations of the shape_S3 class. Imagine that you wanted to create a method that would return TRUE if a shape_S3 object was a square, FALSE if a shape_S3 object was not a square, and NA if the object provided as an argument to the method was not a shape_s3 object. This can be achieved using R's generic methods system. A generic method can return different values based depending on the class of its input. For examplemean() is a generic method that can find the average of a vector of number or it can find the "average day" from a vector of dates.

The following snippet demonstrates this behavior:

```
mean(c(2, 3, 7))
```

```
## [1] 4
```

```
mean(c(as.Date("2016-09-01"), as.Date("2016-09-03")))
```

```
## [1] "2016-09-02"
```

Now let's create a generic method for identifying shape_S3 objects that are squares. The creation of every generic method uses `theUseMethod()` function in the following way with only slight variations:

Let's call this method is_square:

```
is_square <- function(x) UseMethod("is_square")
```

Now we can add the actual function definition for detecting whether or not a shape is a square by specifying is_square.shape_S3.

By putting a dot (.) and then the name of the class after is_squre, we can create a method that associates is_squre with the shape_S3class:

```
is_square.shape_S3 <- function(x){
  length(x$side_lengths) == 4 &&
    x$side_lengths[1] == x$side_lengths[2] &&
    x$side_lengths[2] == x$side_lengths[3] &&
    x$side_lengths[3] == x$side_lengths[4]
```

```
}
is_square(square_4)
```

```
## [1] TRUE
```

```
is_square(triangle_3)
```

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

Seems to be working well! We also want is_square() to return NAwhen its argument is not a shape_S3. We can specifyis_square.default as a last resort if there is not method associated with the object passed to is_square().

```
is_square.default <- function(x){
  NA
}
is_square("square")
```

```
## [1] NA
```

```
is_square(c(1, 1, 1, 1))
```

```
## [1] NA
```

Let's try printing square_4:

```
print(square_4)
```

```
## $side_lengths
## [1] 4 4 4 4
##
## attr(,"class")
## [1] "shape_S3"
```

```
# attr(,"class")
```

Specifiy generic methods for a class

Doesn't that look ugly? Lucky for us `print()` is a generic method, so we can **specify a print method for the shape_S3 class**:

```
print.shape_S3 <- function(x){
  if(length(x$side_lengths) == 3){
    paste("A triangle with side lengths of", x$side_lengths[1],
          x$side_lengths[2], "and", x$side_lengths[3])
  } else if(length(x$side_lengths) == 4) {
    if(is_square(x)){
      paste("A square with four sides of length", x$side_lengths[1])
    } else {
      paste("A quadrilateral with side lengths of", x$side_lengths[1],
            x$side_lengths[2], x$side_lengths[3], "and", x$side_lengths[4])
    }
  } else {
    paste("A shape with", length(x$side_lengths), "slides.")
  }
}
```

```
print(square_4)
```

```
## [1] "A square with four sides of length 4"
```

```
print(triangle_3)
```

```
## [1] "A triangle with side lengths of 3 3 and 3"
```

```
print(shape_s3(c(10, 10, 20, 20, 15)))
```

```
## [1] "A shape with 5 slides."
```

```
print(shape_s3(c(2, 3, 4, 5)))
```

```
## [1] "A quadrilateral with side lengths of 2 3 4 and 5"
```

View all associated methods for a generic function

Since printing an object to the console is one of the most common things to do in R, nearly every class has an assocaited print method! To see all of the methods associated with a generic like `print()` use the `methods()` function:

```
head(methods(print), 10)
```

```
##  [1] "print.acf"     "print.AES"     "print.anova"   "print.aov"
##  [5] "print.aovlist" "print.ar"      "print.Arima"   "print.arima0"
##  [9] "print.AsIs"    "print.aspell"
```

```
head(methods(summary),10)
```

```
##  [1] "summary.aov"                   "summary.aovlist"
##  [3] "summary.aspell"                "summary.check_packages_in_dir"
##  [5] "summary.connection"            "summary.data.frame"
##  [7] "summary.Date"                  "summary.default"
##  [9] "summary.ecdf"                  "summary.factor"
```

One last note on **S3 with regard to inheritance**. In the previous section we discussed how a sub-class can inherit attributes and methods from a super-class. Since you can assign any class to an object in S3, you can specify a super class for an object the same way you would specify a class for an object:

```
class(square_4)
```

```
## [1] "shape_S3"
```

```
class(square_4) <- c("shape_S3", "square")
class(square_4)
```

```
## [1] "shape_S3" "square"
```

To check if an object is a sub-class of a specified class you can use the inherits() function:

```
inherits(square_4, "square")
```

```
## [1] TRUE
```

S4

The S4 system is slightly more restrictive than S3, but it's similar in many ways.

To create a new class in S4 you need to use thesetClass() function. You need to specify two or three arguments for this function: - `Class` which is the name of the class as a string, - `slots`, which is a named list of attributes for the class with the class of those attributes specified, - and **optionally** `contains` which includes the super-class of theyclass you're specifying (if there is a super-class).

Take look at the class definition for a bus_S4 and aparty_bus_S4 below:

```
setClass("bus_S4",
         slots = list(n_seats = "numeric",
                      top_speed = "numeric",
                      current_speed = "numeric",
                      brand = "character"))
setClass("party_bus_S4",
         slots = list(n_subwoofers = "numeric",
                      smoke_machine_on = "logical"),
         contains = "bus_S4")
```

Now that we've created the bus_S4 and the party_bus_S4 classes we can create bus objects using the`new()` function. The `new()` function's arguments are the name of the class and values for each "slot" in our S4 object.

```
my_bus <- new("bus_S4", n_seats = 20, top_speed = 80,
              current_speed = 0, brand = "Volvo")
my_bus
```

```
## An object of class "bus_S4"
## Slot "n_seats":
## [1] 20
##
## Slot "top_speed":
## [1] 80
##
## Slot "current_speed":
## [1] 0
##
## Slot "brand":
## [1] "Volvo"
```

```
my_party_bus <- new("party_bus_S4", n_seats = 10, top_speed = 100,
                    current_speed = 0, brand = "Mercedes-Benz",
                    n_subwoofers = 2, smoke_machine_on = FALSE)
my_party_bus
```

```
## An object of class "party_bus_S4"
## Slot "n_subwoofers":
## [1] 2
##
## Slot "smoke_machine_on":
## [1] FALSE
##
## Slot "n_seats":
## [1] 10
##
## Slot "top_speed":
## [1] 100
##
## Slot "current_speed":
```

```
## [1] 0
##
## Slot "brand":
## [1] "Mercedes-Benz"
```

You can use the @ operator to **access the slots** of an S4 object:

```
my_bus@n_seats
```

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

```
my_party_bus@top_speed
```

```
## [1] 100
```

Implement a new generic Methods

This is essentially the same as using the `$ operator` with a list or an environment.

`S4` classes use a generic method system that is similar to `S3` classes. In order to **implement a new generic method** you need to use:

- the `setGeneric()` function and
- the `standardGeneric()` function in the following way:

```
setGeneric("new_generic", function(x){
  standardGeneric("new_generic")
})
```

```
## [1] "new_generic"
```

Let's create a generic function called `is_bus_moving()` to see if a bus_S4 object is in motion:

```
setGeneric("is_bus_moving", function(x){
  standardGeneric("is_bus_moving")
})
```

```
## [1] "is_bus_moving"
```

Now we need to actually define the function which we can to withsetMethod(). The setMethod() functions takes as arguments the name of the method as a string, the method signature which specifies the class of each argument for the method, and then the function definition of the method:

```
setMethod("is_bus_moving",
          c(x = "bus_S4"),
          function(x){
            x@current_speed > 0
          })
```

```
## [1] "is_bus_moving"
```

```
# Generic function response
is_bus_moving(my_bus)
```

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

```
# Response after reseting speed
my_bus@current_speed <- 1
is_bus_moving(my_bus)
```

```
## [1] TRUE
```

Making use of existing generic methods

In addition to creating your own generic methods, you can also **create a method for your new class from an existing generic**.

First use:

- the `setGeneric()` function with the name of the existing method you want to use with your class,
- and then use the `setMethod()` function like in the previous example.

Let's make a `print()` method for the bus_S4 class:

```
setGeneric("print")
```

```
## [1] "print"
```

```
setMethod("print",
          c(x = "bus_S4"),
          function(x){
            paste("This", x@brand, "bus is traveling at a speed of", x@current_speed)
          })
```

```
## [1] "print"
```

```
print(my_bus)
```

```
## [1] "This Volvo bus is traveling at a speed of 1"
```

```
print(my_party_bus)
```

```
## [1] "This Mercedes-Benz bus is traveling at a speed of 0"
```

Reference Classes

With reference classes we leave the world of R's old object oriented systems and enter the philosophies of other prominent object oriented programming languages.

We can

```
Student <- setRefClass("Student",
                       fields = list(name = "character",
                                     grad_year = "numeric",
                                     credits = "numeric",
                                     id = "character",
                                     courses = "list"),
                       methods = list(
                         hello = function(){
                           paste("Hi! My name is", name)
                         },
                         add_credits = function(n){
                           credits <<- credits + n
                         },
                         get_email = function(){
                           paste0(id, "@jhu.edu")
                         }
                       ))
```

To recap:

- we've created a class definition called Student which defines the student class.

- his class has five fields and three methods.
- To create a Student object use the `new()` method:

```
brooke <- Student$new(name = "Brooke", grad_year = 2019, credits = 40,
                      id = "ba123", courses = list("Ecology", "Calculus III"))
roger <- Student$new(name = "Roger", grad_year = 2020, credits = 10,
                      id = "rp456", courses = list("Puppetry", "Elementary Algebra"))
```

You can **access the fields and methods** of each object using the $operator:

```
brooke$credits
```

```
## [1] 40
```

```
roger$hello()
```

```
## [1] "Hi! My name is Roger"
```

```
roger$get_email()
```

```
## [1] "rp456@jhu.edu"
```

Methods can **change the state of an object**, for instance in the case of the `add_credits()` function:

```
# Displaying the current value of credits for object "brooke"
brooke$credits
```

```
## [1] 40
```

```
# Using the add_credits() instance method to add 4 to the value of credits for instance brooke
brooke$add_credits(4)
# Displaying current credit value
brooke$credits
```

```
## [1] 44
```

Notice that the `add_credits()` method uses the complex assignment operator `(<<-)`. You need to use this operator if you want to modify one of the fields of an object with a method. You'll learn more about this operator in the **Expressions & Environments** section.

`Reference classes` can inherit from other classes by specifying the `contains` argument when they're defined.

Let's create a sub-class of Student called **Grad_Student** which includes a few extra features:

```
Grad_Student <- setRefClass("Grad_Student",
                            contains = "Student",
                            fields = list(thesis_topic = "character"),
                            methods = list(
                              defend = function(){
                                paste0(thesis_topic, ". QED.")
                              }
                            ))

jeff <- Grad_Student$new(name = "Jeff", grad_year = 2021, credits = 8,
                      id = "jl55", courses = list("Fitbit Repair",
                                                  "Advanced Base Graphics"),
                      thesis_topic = "Batch Effects")

jeff$defend()
```

```
## [1] "Batch Effects. QED."
```