Object Oriented Programming Concepts

R - Programming Language

Ashwini Mathur (CSSP)

Learning objectives of this unit are:

Design and Implement a new S3, S4, or reference class with generics and methods

Let's begin the story

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

Object Oriented Systems

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.”

Okay Let's talk about **Object Oriented Principles first a little as a story ...**

There a several key principles in object oriented programming which span across R’s object systems and other programming languages. The first are the ideas of a **class** and an **object**.

Just like the world is full of physical objects, your **programs can be made of objects** as well.

class

A c**lass 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

Methods

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**.

Constructor ..

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.

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

Now Start - S3 Classes

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"
```

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:

```r
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 providied 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 example `mean()` 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 the `UseMethod()` function in the following way with only slight variations:

```
[name of method] <- function(x) UseMethod("[name of method]")
```

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_S3` class:

```
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 `NA` when its argument is not a shape_S3. We can specify `is_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"
```

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"
```

One last note on S3 with regard to inheritance. In the previous section we discussed how a sub-class can inhert 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
```

Now let’s hit the {R Programming} keyboard …

Question : Write a Program for S3 Class / Methods for Polygon.

Time period to Complete : 7 Days (Sunday 22-03-2020 Before 11:59 PM)

Submit you code # **online.classroom.2020@gmail.com**