Statistics 360: Advanced R for Data Science

Lecture 08

Becky Lin

More object-oriented programming in R

- Last time:
  - base objects vs OO objects
  - OOP with S3 in R
- Today:
  - Brief introductions to OOP with R6 and S4
  - Reading: skim chapters 14 and 15 of Advanced R by Wickham

R6

Introduction to R6

- R6 is an "encapculated" OOP system, so methods are bundled with objects, rather than being functions designed to act on objects.
  - R6 methods are called by `object$method()`, rather than `generic.objectclass()`
- R6 objects are implemented using environments and so can be modified in place.
- R6 will look familiar to programmers coming from another language.

```
#install.packages("R6")
library(R6)
```

Defining classes and methods with R6

- Use `R6Class()` to create a class and its methods.
- Two important arguments to `R6Class()` are the classname argument and the public argument.
  - classname is self-explanatory
  - public specifies the methods and fields that are the public interface of the object. Methods access components of the object with `self`.

Example from the text

- Assign the output of R6Class() to a variable with the same name as the class name

```
Accumulator <- R6Class(classname="Accumulator",
  public = list(
                sum = 0,
                add = function(x = 1) {
                      self$sum <- self$sum + x
                      invisible(self)
                      }
                )
)
# Accumulator
x <- Accumulator$new() # create an Accumulator object
x$add(1) # method, after this command: sum=1
x$add(2)$add(3)$add(4) # methods can be "chained"
x$sum # field, sum=10 = 1+2+3+4
```

```
## [1] 10
```

Side-effect methods

- $add() is called for the "side-effect" of modifying the sum
- Side-effect methods should return invisibly. Otherwise the object is printed when the method is called.

```
Accumulator <- R6Class(classname="Accumulator",
  public = list(
                sum = 0,
                add = function(x = 1) {
                      self$sum <- self$sum + x
                      self
                      }
                )
)
# Accumulator
x <- Accumulator$new()
x$add(1)
```

```
## <Accumulator>
##   Public:
##     add: function (x = 1)
##     clone: function (deep = FALSE)
##     sum: 1
```

initialize method

- Will make your class easier to use.
  - initialize is a constructor that over-rides the default new method and allows users to initialize an instance of the class with data values.

```
# version without initialize
Person <- R6Class("Person",
                  list(
                    name = NULL,
                    age = NA
                    ))
emily <- Person$new()
emily$name = "Emily"
emily$age = 25
emily
```

```
## <Person>
##   Public:
##     age: 25
##     clone: function (deep = FALSE)
##     name: Emily
```

```r
# version with initialize
Person <- R6Class("Person",
                  list(
                    name = NULL,
                    age = NA,
  initialize = function(name, age = NA) {
    stopifnot(is.character(name), length(name) == 1)
    stopifnot(is.numeric(age), length(age) == 1)
    self$name <- name
    self$age <- age
    #invisible(self) is not necessary
  }))
emily <- Person$new("Emily", age = 25)
emily
```

```
## <Person>
##   Public:
##     age: 25
##     clone: function (deep = FALSE)
##     initialize: function (name, age = NA)
##     name: Emily
```

validate method

- The above constructor does some checking. More expensive checks should go in a validate method.

```
# version with initialize and validate
Person <- R6Class("Person", list(
  name = NULL,
  age = NA,
  initialize = function(name, age = NA) {
    self$name <- name
    self$age <- age
  },
  validate = function() {
    stopifnot(is.character(self$name), length(self$name) == 1)
    stopifnot(is.numeric(self$age), length(self$age) == 1)
  }
  ))
emily <- Person$new("Emily", age = c(25,35))
try(emily$validate())
```

```
## Error in emily$validate() : length(self$age) == 1 is not TRUE
```

print method

- Add a print method to make printing nicer.

```
# version with initialize and print
Person <- R6Class("Person", list( name = NULL,age = NA,
  initialize = function(name, age = NA) {
    stopifnot(is.character(name), length(name) == 1)
    stopifnot(is.numeric(age), length(age) == 1)
    self$name <- name; self$age <- age},
  print = function(...) {
    cat("Person: \n"); cat("Name:", self$name, "\n")
    cat("Age:", self$age, "\n")
    invisible(self)}
))
emily <- Person$new("Emily", age = 25)
emily
```

```
## Person:
## Name: Emily
## Age: 25
```

Inheritance: Firstly, reset Accumulator

```
Accumulator <- R6Class(classname="Accumulator",
  public = list(
                  sum = 0,
                  add = function(x = 1) {
                            self$sum <- self$sum + x
                            invisible(self)
                            }
                  )
)
```

Inheritance

- Use `inherit` to create a child class that inherits methods and fields from a parent (super) class
- You can add or over-ride methods/fields in the child

```
AccumulatorChatty <- R6Class("AccumulatorChatty",
  inherit = Accumulator,
  public = list(
    add = function(x = 1) {
      cat("Adding ", x, "\n", sep = "")
      super$add(x = x) # use the add in superclass
    }
  )
)
x2 <- AccumulatorChatty$new()
x2$add(10)$add(1)$sum
```

```
## Adding 10
## Adding 1
```

```
## [1] 11
```

class() and names()

- You can use `class()` and `names()` to query an R6 object.

```
class(x2)
```

```
## [1] "AccumulatorChatty" "Accumulator"       "R6"
```

```
names(x2)
```

```
## [1] ".__enclos_env__" "sum"             "clone"           "add"
```

Making copies

- R6 objects are implemented as environments.
  - Objects are modified in place.
  - The usual way of making copies in R with <- does not work:

```
x3 <- x2 # Are we copying x2?
x3$add(100)
```

```
## Adding 100
```

```
x3$sum
```

```
## [1] 111
```

```
x2$sum # !!
```

```
## [1] 111
```

clone

- Make copies with the `$clone()` method.

```
x3 <- x2$clone()
x3$add(-100)
```

```
## Adding -100
```

```
x3$sum
```

```
## [1] 11
```

```
x2$sum
```

```
## [1] 111
```

R6 topics not covered

- Private and active fields (Section 14.3)
- More on unexpected behaviour of R6 classes (Section 14.4)
- R6 *versus* the built-in reference classes (RC) system (Section 14.5)

S4

Introduction to S4

- S4 is a formal functional OOP system with strict rules for creating classes, generics and methods.
- Also has a more advanced implementation of inheritance/dispatch.
- Down-side is that it has a steeper learning curve than S3.
- Terminology: S4 objects have "slots", accessed by `@`.
  - Similar in function to list elements in most S3 classes, which are accessed by `$`.
- S4 is implemented in the `methods` package, which is loaded automatically in every R session.
  - However, the text recommends explicitly loading methods

Creating classes

- Use `setClass` to create a class and the `new()` method to create objects of that class.

```
library(methods)
setClass("Person",
  slots = c(
    name = "character",
    age = "numeric"
  )
)
emily <- new("Person", name = "Emily", age = 25)
```

Class prototype

- In addition to the class and slot names, you should provide a prototype for your class.
  - The prototype specifies default values for the slots

```
setClass("Person",
  slots = c(
    name = "character",
    age = "numeric"
  ),
  prototype = list(
    name = NA_character_,
    age = NA_real_
  )
)
emily <- new("Person", name = "Emily")
str(emily)
```

```
## Formal class 'Person' [package ".GlobalEnv"] with 2 slots
##   ..@ name: chr "Emily"
##   ..@ age : num NA
```

- You can use `is()` to see an S4 object's class, and `@` or `slot()` to access slots.
  - `@` is equivalent to `$` and `slot()` is equivalent to `[[`, and for most purposes they are equivalent to each other.

```
is(emily)
```

```
## [1] "Person"
```

```
emily@name
```

```
## [1] "Emily"
```

```
slot(emily,"name")
```

```
## [1] "Emily"
```

```
emily@name <- "Emily Smith"
emily
```

```
## An object of class "Person"
## Slot "name":
## [1] "Emily Smith"
##
## Slot "age":
## [1] NA
```

Inheritance

- The `contains` argument specifies a parent class to inherit slots and methods from.

```
setClass("Employee",
  contains = "Person",
  slots = c(
    boss = "Person"
  ),
  prototype = list(
    boss = new("Person")
  )
)
emily <- new("Employee",name="Emily",boss=new("Person",name="Catherine"))
is(emily,"Employee")
```

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

```
is(emily,"Person")
```

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

Helpers

- Just as with S3, you should write a user-friendly helper to create objects of your class.
- The helper can perform checks, coerce data to correct types, etc.
- Give the helper the same name as the class

```
Person <- function(name, age = NA) {
  age <- as.double(age)
  new("Person", name = name, age = age)
}
Person("Emily")
```

```
## An object of class "Person"
## Slot "name":
## [1] "Emily"
##
## Slot "age":
## [1] NA
```

Validators

- For more complicated checks, write a validator with `setValidity()`
- `setValidity()` takes a class and a function that returns TRUE if the input is valid, and a character vector describing the problem if not:

```
setValidity("Person", function(object) {
  if (length(object@name) != length(object@age)) {
    "@name and @age must be same length"
  } else {
    TRUE
  }
})
```

```
## Class "Person" [in ".GlobalEnv"]
##
## Slots:
##
## Name:       name        age
## Class: character   numeric
##
## Known Subclasses: "Employee"
```

```r
new("Person",name="Emily",age=25:35)
#Error in validObject(.Object) :
#  invalid class "Person" object:
#        @name and @age must be same length
new("Employee",name=c("Emily","Smith"))
#Error in validObject(.Object) :
#  invalid class "Employee" object:
#       @name and @age must be same length
```

Generics and methods

- Example: Write accessor functions for users to get and set data in your class.
  - Users shouldn't use `@`, and you shouldn't on other developers' classes
  - Write generics with `setGeneric()` and a call to `standardGeneric()`
  - Write methods with `setMethod()`.
  - Setting values has the potential to create invalid objects. Can call `validObject()` to ensure a valid object.

```
# Note: Don't use {} in the function definition of setGeneric.
# get values with a **prefix** function
setGeneric("age", function(x) standardGeneric("age"))
```

```
## [1] "age"
```

```
setMethod("age", "Person", function(x) x@age)
```

```r
# set values with a **replacement** function
setGeneric("age<-", function(x, value)
             standardGeneric("age<-"))
```

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

```r
setMethod("age<-", "Person", function(x, value) {
  x@age <- value
  validObject(x) # check object validity
  x
})
age(emily) <- 25
age(emily)
```

```
## [1] 25
```

```r
try({ age(emily) <- 25:35 })
```

```
## Error in validObject(x) :
##   invalid class "Employee" object: @name and @age must be same lengt
```

Signature

- The signature argument of `setGeneric()` specifies which arguments are used for method dispatch.
  - Default is all arguments.
- The second argument of `setMethod()` is also called signature, and specifies the classes that the method applies to.
- S4 allows for generics and methods to dispatch on multiple classes.
  - Can get quite confusing.
  - See Section 15.5 of text if interested.

```
setGeneric("age<-",function(x,value,...,verbose=TRUE)
             standardGeneric("age<-"),
             signature = "x"
            ) # dispatch on first arg only
```

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

```
setMethod("age<-", "Person",
          function(x,value,...,verbose=TRUE) {
            x@age <- value
            if(verbose) cat("Setting age to",value,"\n")
            x
            }
          )
age(emily) <- 25
```

```
## Setting age to 25
```

show method

- The `show()` method is the S4 equivalent of print.
  - It should have one argument.

```
setMethod("show", "Employee", function(object) {
  cat(is(object)[[1]], "\n",
      "  Name: ", object@name, "\n",
      "  Age:  ", object@age, "\n",
      "  Boss: ", object@boss@name, "\n",
      sep = ""
  )
})
emily
```

```
## Employee
##   Name: Emily
##   Age:  25
##   Boss: Catherine
```

List methods

- Use `methods("generic")` or `methods(class = "class")` to see all methods for a given generic or class.

```
methods("age")
```

```
## [1] age,Person-method
## see '?methods' for accessing help and source code
```

```
methods(class="Employee")
```

```
## [1] age    age<-  coerce show
## see '?methods' for accessing help and source code
```

```
methods(class="Person")
```

```
## [1] age    age<-  coerce
## see '?methods' for accessing help and source code
```

Topics skimmed or not covered.

- Method dispatch, section 15.5
- Interfacing S4 and S3, section 15.6.