

Statistics 360: Advanced R for Data Science

Lecture 7

Brad McNeney

Introduction to object-oriented programming (OOP) in R

- ▶ Reading: Text, OOP Intro, Chapters 12 and 13
- ▶ Topics:
 - ▶ general comments on OOP
 - ▶ terminology
 - ▶ base objects vs OO objects
 - ▶ OOP with “S3” in R

Object-oriented vs functional programming

- ▶ OOP aims to break a problem down into components that are represented by “objects”.
 - ▶ An object contains its data and the functions, or “methods” that can act on that data
- ▶ R is predominantly a functional programming language: we break a problem down into functions.
- ▶ The debate over which style of programming is the best rages on.
 - ▶ Google “functional versus object-oriented programming” for a sample
- ▶ Practical note: algorithms and data structures go hand-in-hand for solving complex problems, and formalizing your data structure as an object is useful.

OOP in R

- ▶ Much of R uses OOP in some form, so if you want to contribute to, extend, or just understand someone else's code you should learn a little OOP.
- ▶ We will discuss three OOP systems: S3, R6 and S4.
 - ▶ S3 is the simplest and most widely used. The text calls it “functional” OOP.
 - ▶ RC and R6 are traditional “encapsulated” OOP systems that looks less familiar to R users but more familiar to programmers from OO languages
 - ▶ S4 is a more formal version of S3
- ▶ Our goal is to learn a little about each style so that we can understand code written by others.
- ▶ For our project we will use S3.
 - ▶ Our MARS function will output an S3 object, and we will write print, summary, plot and other methods to make the interface familiar to R users.

Why use S3?

- ▶ Very minimalist and flexible.
- ▶ Widely used, so others can understand your code.
- ▶ OOP can evolve as you work.
 - ▶ Just like R is good for prototyping functional algorithms, S3 is good for prototyping OOP methods.
 - ▶ If your code needs more structure (e.g., you are starting to open it up to collaborators), you can formalize your OOP then.
- ▶ It “looks like R”

Terminology: polymorphism and encapsulation

- ▶ **polymorphism:** As we've seen, functions like `plot` are generic and behave differently when given different inputs; this is called polymorphism.
- ▶ **encapsulation:** We hide the details of an object behind an interface.
 - ▶ Encapsulated OOP formally bundles data and methods used to set and get data values; methods are called as `object.method(args)`.
 - ▶ Functional OOP provides “generic” functions that the user *should* use to get and set data; generic functions are called as `generic(object,args)` and “method dispatch” is used to find the correct method (more on this later).
 - ▶ Encapsulation allows the developer to change the implementation of the object without breaking other code: just change the relevant methods for getting and setting data.

Example of functional OOP

```
data(mtcars)
ff <- lm(mpg~disp,data=mtcars)
class(ff)
```

```
## [1] "lm"
```

```
# names(ff)
# ff$residuals
# residuals(ff)
# residuals
# summary(ff)
```


Terminology, continued

- ▶ We have been using the terms **class** and **method**.
- ▶ Other important terms:
 - ▶ **fields:** are the data of the class
 - ▶ **inheritance:** Classes can be organized in a hierarchy that we search for an appropriate method. If a method does not exist for a child, or sub-class and we use the method from the parent, or super-class, then the child is said to inherit behaviour from the parent.
 - ▶ **method dispatch** is the process of finding an appropriate method for a given class

Base objects vs OO objects

- ▶ We have been using “object” to describe data and functions in R in general.
- ▶ Now distinguish between “base” objects, such as numeric vectors, and OO objects that have a class attribute.

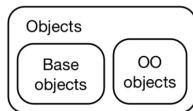


Figure 1: Objects

```
x <- 1:4  
attr(x,"class") # compare with class(x) -- misleading
```

```
## NULL
```

```
attr(mtcars,"class")
```

```
## [1] "data.frame"
```

```
attr(ff,"class")
```

```
## [1] "lm"
```

```
is.object(x)
```

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

```
is.object(mtcars)
```

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

```
is.object(ff)
```

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

Base types

- ▶ Recall that base objects have a type that you can discover with `typeof()`.
- ▶ There are 25 base types.
- ▶ These types describe the underlying implementation of the base object in memory, and functions that behave differently for different base types are coded with switch statements.
- ▶ See the text, section 12.3 for more details on base object types.

OOP with S3

- ▶ S3 is an informal OO system and the most commonly-used.
 - ▶ E.g, it is the only OO system used in base R and the R stats package.
- ▶ Without strict rules, you have a lot of freedom, but can also write bad code.
- ▶ We will discuss conventions for creating useful classes and methods
- ▶ We will use the `sloop` package recommended by the text to query objects about their class and available methods.

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

S3 classes

- ▶ An S3 class is a base type with a class attribute

```
f <- factor(c("cat", "dog", "mouse"))  
typeof(f)
```

```
## [1] "integer"
```

```
attributes(f) # see also class(f) and inherits(f, "factor")
```

```
## $levels
```

```
## [1] "cat"    "dog"    "mouse"
```

```
##
```

```
## $class
```

```
## [1] "factor"
```

```
otype(f) # from sloop
```

```
## [1] "S3"
```

```
s3_class(f) # from sloop
```

```
## [1] "factor"
```

Creating your own class

- ▶ Use `class()` to set the class after the object has been created, or use `structure()`:

```
new_node <- function(data, childl=NULL, childr=NULL){  
  structure(list(data=data, childl=childl, childr=childr),  
            class="node")  
}  
nn <- new_node(data=NULL) # Note: data should be a region object  
s3_class(nn)
```

```
## [1] "node"
```

Removing the class attribute

- ▶ We can simply remove the class with `attributes(f)$class <- NULL` but it is better to use `unclass()`.

```
print(unclass(f))
```

```
## [1] 1 2 3
```

```
## attr(,"levels")
```

```
## [1] "cat" "dog" "mouse"
```

```
otype(unclass(f))
```

```
## [1] "base"
```


Class conventions

- ▶ No rules, but the text suggests a few conventions.
- ▶ Naming: Any string is OK, but stay away from `.`, which is the separator between generic and class names in naming methods.
- ▶ Constructor: Make a function named `new_myclass()` to create an object with the correct structure.
- ▶ Validator: Make a function named `validate_myclass()` that checks that the object's data makes sense, stops if not, and otherwise returns the object.
- ▶ Helper: Make a function named `myclass()` that users can use to create instances of the class.
- ▶ Exercise (see week 7 exercises): Create validator and helper functions for the node class example on the previous slide.

Constructors

- ▶ Text: The constructor should
 - ▶ Be called `new_myclass()`.
 - ▶ Have one argument for the base object, and one for each attribute.
 - ▶ Check the type of the base object and the types of each attribute.
- ▶ Note: I often write constructors whose base object is a list, and my constructor has separate arguments for each list element.
 - ▶ See the `new_node()` function, for example
 - ▶ This goes against the second of the above conventions.

Example constructor

- From the text: Make a constructor for the S3 class difftime

```
new_difftime <- function(x = double(), units = "secs") {  
  stopifnot(is.double(x))  
  units <- match.arg(units, c("secs", "mins", "hours", "days", "weeks"))  
  
  structure(x,  
    class = "difftime",  
    units = units # set a "units" attribute  
  )  
}  
new_difftime(c(1, 10, 3600), "secs")
```

```
## Time differences in secs  
## [1]      1     10 3600
```

```
new_difftime(52, "weeks")
```

```
## Time difference of 52 weeks
```

```
try(new_difftime(1, "eon"))
```

```
## Error in match.arg(units, c("secs", "mins", "hours", "days", "weeks")) :  
##   'arg' should be one of "secs", "mins", "hours", "days", "weeks"
```

Notes on constructors

- ▶ Think of the constructor as a function to be used by you or other knowledgeable users.
 - ▶ Don't need extensive checking
- ▶ Time-consuming checks should go in the validator ...

Validator

- ▶ Write a validator, `validate_myclass()` if checking the validity of the object's data may be computationally expensive.

```
validate_difftime <- function(x) {  
  # if(bad_thing(x)) stop("Bad thing has happened")  
  x # return object if it passes all checks  
}
```

Helper

- ▶ This is for ordinary users.
- ▶ Should have the same name as the class.
- ▶ Should have as many defaults as practical to make it easy to use.
- ▶ Should call the validator, if one exists.

```
difftime <- function(x = double(), units = "secs") {  
  x <- as.double(x) # try coercing input to required double  
  x <- validate_diffime(x) # validate  
  new_diffime(x, units = units) # call constructor  
}
```

S3 generic functions and methods

- ▶ A generic function, like `print`, defines an interface (arguments) and finds an appropriate method
 - ▶ the method is an implementation specific to the object class
 - ▶ finding an appropriate method is method dispatch

```
print
```

```
## function (x, ...)  
## UseMethod("print")  
## <bytecode: 0x1494db918>  
## <environment: namespace:base>
```

```
ftype(print)
```

```
## [1] "S3"      "generic"
```

```
ftype(print.factor)
```

```
## [1] "S3"      "method"
```

Method dispatch: UseMethod()

- ▶ In simple cases, UseMethod() looks for `generic.class()`, and falls back on `generic.default()`.
 - ▶ If neither exist, it throws an error.
- ▶ When a class inherits from a parent class, the search gets more complicated.

Example: print methods

- ▶ When you type the name of an object in the R console you invoke print.
- ▶ Different print methods exist for different classes of objects.
 - ▶ How many?

```
s3_methods_generic("print") # a lot!
```

```
## # A tibble: 214 x 4
##   generic class   visible source
##   <chr>   <chr>   <lgl>   <chr>
## 1 print  acf      FALSE   registered S3method
## 2 print  AES      FALSE   registered S3method
## 3 print  anova    FALSE   registered S3method
## 4 print  aov      FALSE   registered S3method
## 5 print  aovlist  FALSE   registered S3method
## 6 print  ar       FALSE   registered S3method
## 7 print  Arima    FALSE   registered S3method
## 8 print  arima0   FALSE   registered S3method
## 9 print  AsIs     TRUE    base
## 10 print aspell   FALSE   registered S3method
## # ... with 204 more rows
```

Example: print.factor

- ▶ Most S3 methods are not exported from the packages in which they are defined, but you can view them with sloop.

```
s3_get_method("print.factor")
```

```
## function (x, quote = FALSE, max.levels = NULL, width = getOption("width"),
##   ...)
## {
##   ord <- is.ordered(x)
##   if (length(x) == 0L)
##     cat(if (ord)
##       "ordered"
##       else "factor", "(0)\n", sep = "")
##   else {
##     xx <- character(length(x))
##     xx[] <- as.character(x)
##     keepAttrs <- setdiff(names(attributes(x)), c("levels",
##       "class"))
##     attributes(xx)[keepAttrs] <- attributes(x)[keepAttrs]
##     print(xx, quote = quote, ...)
##   }
##   maxl <- if (is.null(max.levels))
##     TRUE
##   else max.levels
##   if (maxl) {
##     n <- length(lev <- encodeString(levels(x), quote = ifelse(quote,
##       "\"", "")))
##     colsep <- if (ord)
##       "< "
##     else " "
##     T0 <- "Levels: "
##     if (is.logical(maxl))
##       maxl <- {
##         width <- width - (nchar(T0, "w") + 3L + 1L +
```

Writing methods when there is a generic

- ▶ Just write a function with name `generic.class`
 - ▶ See our `print.region()` method from lab 3.
- ▶ The method should have the same arguments as the generic.
 - ▶ In the case of `print()` there is just one required argument, the object to be printed.

Writing a generic

- ▶ Just need a call to UseMethod()

```
plot_regions <- function(x,...) UseMethod("plot_regions")
plot_regions.tree <- function(tree){
  # set up empty plot
  plot(tree$data$x[,1],tree$data$x[,2],xlab="X1",ylab="X2")
  plot_regions.node(tree$childl)
  plot_regions.node(tree$childr)
}
# add plot_regions.node(), recpart() then test
```

Using inheritance

- ▶ Our MARS objects will contain the output of the final call to `lm()`.
- ▶ Can make `lm` the parent class of our MARS objects.
 - ▶ Toy example:

```
new_mars <- function(formula,data) {  
  ff <- lm(formula,data)  
  structure(c(ff,list(ID="Hi, I'm a MARS object")),  
            class=c("mars",class(ff)))  
}  
mm <- new_mars(mpg~cyl,data=mtcars)  
s3_dispatch(print(mm))  
  
##      print.mars  
## => print.lm  
## * print.default
```

Further reading

- ▶ If you are interested in reading more about S3 classes on your own, see chapter 13 of the text.
- ▶ Topics we skipped or skimmed:
 - ▶ Object styles (section 13.5)
 - ▶ Inheritance: `NextMethod()` and subclassing (section 13.6)
 - ▶ Dispatch details (section 13.7)