Chapter_4

Henry Bourne

2022-10-24

Functional and Object-Oriented Programming

In this chapter we will go over how to write in both the functional and object oriented programming paradigms in R. First we will look at how to programme functionally.

Functional Programming

R supports many functional programming features, perhaps the most important is that is has **first-class functions**: a programming language has first-class functions if you can define a function, functions can be arguments to other functions, functions can be returned by functions and functions can be stored in data structures.

When functional programming it is good to avoid writing functions that modify the global programme state. A **pure function** is a function which given the same arguments returns the same output and has no side affects. Although it is unrealistic that all your functions will be pure, for example functions that generate pseudo-random numbers will not be pure.

In R we have that functions only have access to variables defined in their environment. If in the environment that a function is declared there are free variables (variables used in the function that haven't been assigned a value), then the environment binds the free variables and the function is a **closure**. So if there are free variables we get back a function and we can then pass to the function the arguments which we would like it to use for the free variables. For example:

```
f <- function(x){
  add_y <- function(y){
    x + y
  }
}
add_10 <- f(10)
add_10(5)
```

```
## [1] 15
```

One thing to take especial note of is that R uses lazy evaluation. **Lazy evaluation** means that R doesn't evaluate an expression until it's value is needed. So, in the example above R doesn't evaluate the expression until we've given *add_10()* its argument and expect a value returned, ie. it doesn't evaluate the expression when we call f. This saves allot of computation, however, we must be careful as if we change variables along the way we must keep in mind that R will only evaluate when a value is needed (and therefore will use the most recently defined (current) values for variables).

Object-Oriented Programming (OOP)

Now we will look at how to do OOP in R. Note that in R this isn't very straightforward and there are actually multiple different ways in which we can programme in an object-oriented way in R. The key idea of OOP is **polymorphism** which essentially means that a single symbol may refer to different types. In

OOP we have that data and methods that operate on the data are bundled or **encapsulated** together in an object. This means that we can hide values or the state of a structured data object inside a class preventing direct access to them. The state of an object is defined by its **fields**, we can then perform operations on the state by defining **methods**. Many OOP implementations allow a class to **inherit** all the fields and methods from another class. We can use the **Unified Modelling Language (UML)** to graphically display the hierarchical structures of our classes (this is called a class diagram). We can also use OOP to create general purpose solutions to potential problems, allowing us to quickly perform a desired function without having to rewrite code, these general, reusable solutions are called design **patterns**.

In base R there are three ways of carrying out OOP. The first is S3 which is the least rigorous and can be thought of as functional OOP. Second is S4 which requires formal definitions. And lastly we have Reference Classes which implement properly encapsulated OOP, Reference Classes are also **mutable** (can be modified in-place). We start by describing S3...

S3

An **S3 object** is a base object with attribute class set to the class name. Everything in R is an object, however not all of these objects are object-oriented (OO) objects, **base objects** are objects that don't have a class attribute (and hence aren't OO objects). An **S3 object** is simply a base object but with a class attribute (set to the class name). We use the *class()* function to set the class attribute of an object and hence create an S3 object, we can also use the *unclass()* function to strip the class name attribute and turn it back into a non OO object. It is good practice to have a **constructor** function that a user can call to initialize an object correctly and then return it, like how you would create an object in other languages such as python. It's also recommended to include a **helper** function which is there to correct common mistakes on the fly.

We can add a method to an S3 object by assigning a function to the following name ., note that this is only for implementing methods for which there exists a generic function of the same name such as plot or print. Also note that the method must take the same arguments as the generic. If we want to we can create a new generic function using *<- function( ){ UseMethod("") }* then assign a method with the new generic method name to our S3 class. We can also inherit in S3 by setting the class attribute to a vector of class names, to do this the child class will have a vector containing the names of it and its parent classes.

S4

Compared to S3, S4 has a more formal approach to OOP. In S4 we have to define classes in a more formal way, the advantage of this is that it makes working with S3 more clear and allows for build-in integrity checks. To use S4 we need to make sure we have the "methods" package installed, after we've done this we can then define a class as follows:

```
library(methods)
setClass("class_name",
   slots = c(
     slot_1  = "numeric",
     slot_2 = "logical",
     slot_3  = "character"
     )
)
```

Here we have defined a class with the name "class_name" and three attributes labelled slot_1, slot_2, slot_3 with types numeric, logical and character, ie. to create a class we call *setClass()*, passing it two arguments, the first being the class name and the second a vector of **slots** (same as fields or methods in other OOP languages). After defining the slots we need to define the methods, we will now show an example of a special method for a class which is the initialization method (the method we call to create an object from the class):

```
setMethod("initialize", "class_name",
  function(.Object, slot_1, slot_2) {
    .Object@slot_1  <- slot_1
```

```
    .Object@slot_2 <- slot_2

    if(slot_2){
      x <- as.character(slot_1)
    }
    else{
      x <- "no number"
    }
    .Object@slot_3 <- x
    return(.Object)
  }
)
```

To create a method therefore, we call *setMethod*. The first argument we pass to it is the name of the generic function we want to implement a method for, here we use "initialize" which is a non-standard generic function which is called within the function *new()* (which constructs a new object of the class corresponding to the class name for which it is a method). The second argument is the *signature*, which is the classes that the arguments of the function need for this implementation of the function to be able to run. The third argument is the function for the method, here the function has .Object as the first argument, this is because to create a new object we call *new()* with the first argument set to the class name, this then calls initialize, passing the object prototype for the class to the .Object argument. The next two arguments specify that only two more arguments are to be given to *new()* when we are creating an object, here we say that two arguments slot_1 and slot_2 should be given. An example of how we would create an object is:

```
new("class_name", 1, TRUE)
```

```
## An object of class "class_name"
## Slot "slot_1":
## [1] 1
##
## Slot "slot_2":
## [1] TRUE
##
## Slot "slot_3":
## [1] "1"
```

In the function itself we can set the values of slots by writing *.Object@* <- , to achieve proper encapsulation the @ operator should only be used within method definitions. To allow the user to change or fetch slot values we should create getter and setter methods. An example of how we would define a getter and setter method for slot_1 is:

```
setGeneric("slot_1", function(x) standardGeneric("slot_1"))
```

```
## [1] "slot_1"
```

```
setGeneric("slot_1<-", function(x, value) standardGeneric("slot_1<-"))
```

```
## [1] "slot_1<-"
```

```
setMethod("slot_1", "class_name", function(x) x@regressor)
setMethod("slot_1<-", "class_name",
  function(x, value) {
    x <- initialize(x, slot_1 = value, slot_2 = x@slot_2)
    validObject(x)
    return(x)
  }
)
```

In the final line of the function we return the object, hence when we call *new()* we get given back an instantitation of the class with all its slots set appropriately. Note that again in S4 methods must be implementations of a generic function, to create a new generic function we can use the command *setGeneric()*. It is often useful to check in our methods that we have a valid object, to do this we can use the command *validObject()*, this is one of the build-in mechanisms we mentioned earlier. By default it checks that all slots are present and of the correct type, we can also add validity check by using the function *setValidity()* and passing it the class name for which we want to add a validity check and a method which carries out the validity check.

We can also have relationships between objects in S4. For example two individual objects can be related to each other, to model this relation we create a slot in each of the classes for which their objects will be related and make the type of the slot the class of the related object. When all the objects of one class are related to all of the objects of another class (ie. one class inherits or is a more general version of another class) then we can model this by adding the *contains* argument to the call of setClass and setting it to a vector of the classes from which it should inherit. This then means that the child class will have all the slots the parent class has and methods. In regards to methods if for a particular method the child class has an implementation of this method then this will be called, if it doesn't then the parent class will be checked for this method and used if it has it, if not then the grandparent class will be checked and so on.

Reference Classes

The final way of doing OOP in R that we will discuss is reference classes. Reference classes provide a way of conducting OOP in R with a higher degree of encapsulation than with S3 and S4, it also allows for modify-in-place (memory changed directly) as opposed to copy-on-modify (creates a copy in memory with modification) which is used most of the time in R. We will explain how reference classes work within the context of an example, what we will do is create a class that can carry out cross-validation for the "stattools" package. First we install the stattools package:

```
library(stattools)
# Can use "devtools::install_github("h-aze/compass_yr1", subdir = "/labs/stattools")" if not already do
```

Currently the cross_validation function in "stattools" is implemented as so:

```
regr_cross_val <- function(D, y, RM=LLS, k=10, ...){

  # We randomly shuffle the indices of the rows and using these randomized indices shuffle the rows of
  if(is.vector(D)){
    ind <- sample(length(D))
    D_dash <- D[ind]
    y_dash <- y[ind]
  }
  else if(is.array(D)){
    ind <- sample(nrow(D))
    D_dash <- D[ind,]
    y_dash <- y[ind]
  }
  else{
    ind <- sample(nrow(D))
    D_dash <- D[paste(ind),]
    y_dash <- y[ind]
  }

  # We now create a list which indexes the rows in our dataset, we will use this to select groups of ce
  #   Hence what we have effectively done here is partition the dataset (randomly - as we shuffled the
  #   We split the dataset into 10 groups as we will be performing 10-fold cross validation.
  if(is.vector(D)){
```

```
    subsets <- cut(seq(1,length(D)), breaks = k, labels = FALSE)
  }
  else{
    subsets <- cut(seq(1,nrow(D)), breaks = k, labels = FALSE)
  }

  # We create a vector to store our cross-val errors
  errors <- c()

  # Loop for carrying out 10-fold cross-val
  #TODO: be able to adjust the number of folds
  for(i in 1:k){

    # We segment our data into testing, D.test, and training, D.train, datasets
    testIndexes <- which(subsets==i,arr.ind=TRUE)
    if(is.vector(D)){
      D.test <- D_dash[testIndexes]
      D.train <- D_dash[-testIndexes]
    }
    else{
      D.test <- D_dash[testIndexes, ,drop=FALSE]
      D.train <- D_dash[-testIndexes, ,drop=FALSE]
    }

    # We get the model matrix and predictor variables for the training data
    # and then we find the LS estimator
    X.train <- model_matrix(D.train)
    y.train <- y_dash[-testIndexes]
    w <- RM(X.train, y.train,...)

    # We get the model matrix and predictor variables for the testing data
    # and then we calculate the least squares testing error
    X.test <- model_matrix(D.test)
    y.test <- y_dash[testIndexes]
    test.error <- norm(y.test - X.test %*% w, type="2")**2
    # We add the testing error to our error vector
    errors[i] <- test.error

  }

  # We find and return the cross-val error
  error.CV <- sum(errors)/k
  error.CV
}
```

The problem with this function is that it can be a hastle to use. If we want to carry out cross-validation multiple times with a specific regression method we must specify it each time and if the regression method has hyper-parameters then we must first specify the regression method, RM, we want to use and the k we want to use, even if we are happy with the default values. Further we may sometimes want more information from this function, such as the estimates it found, or we may want to change the cost function being used. By writing a class for this function we will be able to solve all of the above problems and add extra functionality, whilst making it easier for the user.

We have already made sure that we have the methods package in this RMarkdown script, so now we can

create the class:

```
CrossValidation <- setRefClass("CrossValidation",
                               fields=c( data="numeric",
                                         target="numeric",
                                         k="integer",
                                         Regr_method="ANY",
                                         Regr_method.name="character",
                                         E_fun = "ANY",
                                         E_fun.name = "character"
                                         )
                               )
```

Here we have created a class called CrossValidation and defined the fields it will have along with their names. Note that in reference classes they are called fields and not slots like in S4. Now we would like to define some methods for the class:

```
#need to do functional closure regr_methods
CrossValidation$methods(
  initialize = function(data, target, k=as.integer(10), Regr_method=LLS, E_fun=E_l2 ) {
    .self$data <- data
    .self$target <- target
    .self$k <- k
    .self$Regr_method <- Regr_method
    .self$Regr_method.name <- as.character(substitute(Regr_method))
    .self$E_fun <- E_fun
    .self$E_fun.name <-as.character(substitute(E_fun))
  },

  #TODO: track certain things
  # Carries out k-fold cross-validation for a regression problem
  regr_cv = function(){
    # We randomly shuffle the indices of the rows and using these randomized indices shuffle the rows o
    if(is.vector(.self$data)){
      ind <- sample(length(.self$data))
      D_dash <- .self$data[ind]
      y_dash <- .self$target[ind]
    }
    else if(is.array(.self$data)){
      ind <- sample(nrow(.self$data))
      D_dash <- .self$data[ind,]
      y_dash <- .self$target[ind]
    }
    else{
      ind <- sample(nrow(.self$data))
      D_dash <- .self$data[paste(ind),]
      y_dash <- .self$target[ind]
    }

    # We now create a list which indexes the rows in our dataset, we will use this to select groups of
    if(is.vector(.self$data)){
      subsets <- cut(seq(1,length(.self$data)), breaks = .self$k, labels = FALSE)
    }
    else{
      subsets <- cut(seq(1,nrow(.self$data)), breaks = .self$k, labels = FALSE)
```

```
  }

  # We create a vector to store our cross-val errors
  errors <- c()

  # Loop for carrying out k-fold cross-val
  for(i in 1:.self$k){
    # We segment our data into testing, D.test, and training, D.train, datasets
    testIndexes <- which(subsets==i,arr.ind=TRUE)
    if(is.vector(.self$data)){
      D.test <- D_dash[testIndexes]
      D.train <- D_dash[-testIndexes]
    }
    else{
      D.test <- D_dash[testIndexes, ,drop=FALSE]
      D.train <- D_dash[-testIndexes, ,drop=FALSE]
    }

    # We get the model matrix and predictor variables for the training data and then we find the esti
    X.train <- model_matrix(D.train)
    y.train <- y_dash[-testIndexes]
    w <- .self$Regr_method(X.train, y.train)

    # We get the model matrix and predictor variables for the testing data and then we calculate the
    X.test <- model_matrix(D.test)
    y.test <- y_dash[testIndexes]
    test.error <- .self$E_fun(y.test, X.test %*% w)

    # We add the testing error to our error vector
    errors[i] <- test.error

  }

  # We find and return the cross-val error
  error.CV <- sum(errors)/.self$k
  error.CV
},

show = function() {
  cat("head(data)   =", head(.self$data), "\n", sep=" ")
  cat("head(target) =", head(.self$target), "\n", sep=" ")
  cat("k =", .self$k, "\n", sep=" ")
  cat("Regr_method =", .self$Regr_method.name, "\n", sep=" ")
  cat("E_fun =", .self$E_fun.name, "\n", sep=" ")
},
getResult = function() {
  if (!flag) {
    .self$doComputation()
  }
  return(result)
},
setData = function(value) {
  .self$initialize(data = value)
```

```
  }
)
```

Now we will test out this class, first we create some data to test it on and initialize the class,

```
x <- runif(100,0,10)
y <- 2*x -1 + rnorm(100)
cv <- CrossValidation$new(data=x,  target=y) ;cv
```

```
## head(data)   = 3.588997 1.942522 0.7969842 1.167627 2.134124 3.98403
## head(target) = 5.223201 2.811287 0.03603675 2.76864 4.174771 5.536561
## k = 10
## Regr_method = LLS
## E_fun = E_l2
```

Now we run cross validation on our toy data:

```
cv$regr_cv()
```

```
## [1] 9.747326
```

means we can also add a classification method later