Data Programming with R

Isabella Gollini

Lecture 8 - Object oriented programming

Object oriented programming

- What is object-oriented programming?
- S3 classes
  - Writing S3 classes
  - What is inheritance
- Extended example
- S4 classes
- Managing objects

Object oriented programming

- Everything we use in R (functions, vectors, data frames, etc) is an **object**.
- There are several important features of OOP that R promotes:
  - **Encapsulation** - data items are packaged into one class instance and makes keeping track of everything easier.
  - **Polymorphism** - the same function will have different actions on different classes.
  - **Inheritance** - objects of one class can inherit characteristics of another class.
- R mainly uses two types of classes: **S3** and **S4**. In this lecture we'll study them in detail and go through the differences

S3 classes

- An S3 class consists of a **list** with a **class name** attribute (e.g. `lm`) and **dispatch capability**.
- The **dispatch capability** means that you can make use of generic functions (e.g. `print` or `summary`).
- S4 classes (such as the `pixmap` class we met in lecture 2) were included later to add safety.
- In S4 classes you can't accidentally access a class component (e.g. a tag in a list) that you have not already defined.

S3 generic functions

We have met a number of functions that work on vectors, lists, matrices, etc: `print`, `plot`, `summary`.

These are known as **generic functions**, in that they act differently for different types of objects given to them.

```
y <- list(tag1 = 2, tag2 = c(2, 3), tag3 = 1:10)
print(y)
```

```
## $tag1
## [1] 2
##
## $tag2
## [1] 2 3
##
```

```
## $tag3
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
mod <- lm(Fertility ~ ., data = swiss)
print(mod)
```

```
##
## Call:
## lm(formula = Fertility ~ ., data = swiss)
##
## Coefficients:
##      (Intercept)       Agriculture       Examination         Education
##          66.9152           -0.1721           -0.2580           -0.8709
##         Catholic  Infant.Mortality
##           0.1041            1.0770
```

S3 generic functions

The reason R acts differently is that it looks at the class of the objects:

```
class(y)
```

```
## [1] "list"
```

```
class(mod)
```

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

When R sees an object of non standard class it looks for a special print method, in this case `print.lm`. You can look at the code for the two different methods:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x7ff5dc419e80>
## <environment: namespace:base>
```

```
stats:::print.lm
```

```
## function (x, digits = max(3L, getOption("digits") - 3L), ...)
## {
##     cat("\nCall:\n", paste(deparse(x$call), sep = "\n", collapse = "\n"),
##         "\n\n", sep = "")
##     if (length(coef(x))) {
##         cat("Coefficients:\n")
##         print.default(format(coef(x), digits = digits), print.gap = 2L,
##             quote = FALSE)
##     }
##     else cat("No coefficients\n")
##     cat("\n")
##     invisible(x)
## }
## <bytecode: 0x7ff5d9cebd38>
## <environment: namespace:stats>
```

- See the screencast for a more detailed discussion of classes and generic functions.

Removing the class

You can see what the object `mod` really looks like underneath with `unclass`:

```
unclass(mod)
```

`unclass(mod)` just returns a (very long) list.

```
names(mod)
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
```

The `print.lm` function just takes elements of that list and prints out the most important bits.

```
stats:::print.lm(mod)
```

```
##
## Call:
## lm(formula = Fertility ~ ., data = swiss)
##
## Coefficients:
##      (Intercept)       Agriculture       Examination         Education
##          66.9152           -0.1721           -0.2580           -0.8709
##         Catholic  Infant.Mortality
##           0.1041            1.0770
```

This saves the user from seeing lots of unimportant material that may not interest them.

What generic functions are available?

You can find all the versions of generic functions with `methods`.

Methods with a `*` are non visible functions. These are functions that are not meant for use by users but by other functions in a package. You can use them yourself by typing e.g. `getAnywhere(print.acf)`.

```
methods(print)[1:10]
```

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

Writing S3 classes

To create an object of S3 class we simply create a `list` and give it a `class`. We can then write generic functions for that particular class.

```
j <- list(name = 'Joe', salary = 55000, union = TRUE)
class(j) <- 'employee'
j # or print(j)
```

```
## $name
## [1] "Joe"
##
## $salary
## [1] 55000
##
```

```
## $union
## [1] TRUE
##
## attr(,"class")
## [1] "employee"
```

Writing S3 classes

We can now write a new `print` method:

```r
print.employee <- function(wkr) {
  cat(wkr$name, '\n')
  cat('salary', wkr$salary, '\n')
  cat('union member', wkr$union, '\n')
}
j # or print.employee(j) or print(j)
```

```
## Joe
## salary 55000
## union member TRUE
```

This is far neater than the original `print` command.

Using inheritance

Objects can have more than one class. The generic functions will then look for methods by looping through the classes:

```r
k <- list(name = 'Kate', salary = 68000, union = F, hrs_this_month = 2)
class(k) <- c('hrly_employee', 'employee')
k
```

```
## Kate
## salary 68000
## union member FALSE
```

Here, the `print` method looked for `print.hrly_employee` but couldn't find it so used `print.employee` instead.

Extended example

Polynomial regression

Polynomial regression

SC 2

- A commonly used statistical method for one response and one explanatory variable is polynomial regression:

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \ldots + \beta_p x_i^p + \epsilon_i$$

- One method for choosing the degree of the fit (i.e. the value of p) is to use cross-validation (CV).
- There are different versions of CV: we will use leave-one-out CV (LOO-CV): we drop each observation and re-fit the model.
- We will choose the value of p by finding the value which minimises the LOO-CV squared error.

- We will create a new class `polyreg` and a `summary` method for this object.
- See the screencast for a walkthrough of this section.

Function `lv_one_out`

First function: fit a leave one out model and create predicted values for the left out observation.

```
lv_one_out <- function(y, xmat) { # xmat matrix containing x, x^2,..., x^p
  n <- length(y) # number of observations
  predy <- vector(length = n) # initilise the vector of the predicted y
  for(i in 1:n) {
    lmo <- lm(y[-i] ~ xmat[-i,,drop = FALSE]) # apply the regression to all the observation but i
    betahat <- as.vector(lmo$coef) # store the coefficients in the vector beta
    predy[i] <- betahat %*% c(1, xmat[i,]) # predict y_i
}
  return(predy)
}
```

Function `polyfit`.

Second function: fit a regression model for each degree and calculate LOO-CV fitted values from `lv_one_out`:

```
polyfit <- function(y, x, maxdeg) {
  n <- length(y)
  # create Xmat matrix containing x, x^2,..., x^maxdeg
  Xmat <- sweep(matrix(rep(x, maxdeg), nrow = n, ncol = maxdeg), 2, 1:maxdeg, '^')
  # create a list of class polyreg that will contain the output
  lmout <- list()
  class(lmout) <- 'polyreg'
  # fit different polynomial regressions, from degree 1 (linear regression) to degree maxdegree
  for(i in 1:maxdeg) {
    lmo <- lm(y ~ Xmat[, 1:i, drop = FALSE])
    # add the results of the cross validation in a new tab of lmo
    lmo$cv.fitted.values <- lv_one_out(y, Xmat[, 1:i, drop = FALSE])
    lmout[[i]] <- lmo # store the results of this regression in a tab of lmout
  }
  # store also the data in the output
  lmout$x <- x
  lmout$y <- y
  return(lmout)
}
```

Using the `polyfit` function

```
n <- 60
x <- (1:n) / n
y <- rnorm(n, sin(3 * pi / 2 * x) + x^2, sd = 0.5)
maxdeg <- 6
lmo <- polyfit(y, x, maxdeg)
summary(lmo)
```

```
##   Length Class  Mode
```

```
##    13     lm     list
##    13     lm     list
##    13     lm     list
##    13     lm     list
##    13     lm     list
##    13     lm     list
## x 60     -none- numeric
## y 60     -none- numeric
```

Function `summary.polyreg`

Final function: write a `summary` method for the `polyreg` class:

```
summary.polyreg <- function(lmo.obj) {
    maxdeg <- length(lmo.obj) - 2 # lmo.obj contains the outputs and x and y
    n <- length(lmo.obj$y)
    tbl <- matrix(nrow = maxdeg, ncol = 1) # table for the summary
    colnames(tbl) <- 'MSPE'
    rownames(tbl) <- paste("degree", 1:maxdeg, sep = "_")
    for(i in 1:maxdeg) {
      curr.obj <- lmo.obj[[i]]
      errs <- lmo.obj$y - curr.obj$cv.fitted.values # absolute error in predicting y
      spe <- crossprod(errs, errs) # sum of squares of the vector errs
      tbl[i, 1] <- spe / n
    }
    cat('Mean squared predictions errors, by degree \n')
    print(tbl)
}
```

Using the functions

Repeat the function `polyfit` with the new `summary` method:

```
lmo <- polyfit(y, x, maxdeg)
summary(lmo)
```

```
## Mean squared predictions errors, by degree
##               MSPE
## degree_1 0.3063560
## degree_2 0.2784277
## degree_3 0.2484512
## degree_4 0.2523348
## degree_5 0.2590406
## degree_6 0.2638543
```

S4 classes

- Problems with S3
- An example
- Printing S4 classes
- Writing generic functions
- S3 vs S4

Problems with S3

- S3 classes do not provide much safety if we make a mistake.
- For our example on employees (slide 10), we might:
  - Forget to enter a field.
  - Mis-spell one of the fields.
  - Accidentally give a different object the class `employee`.
- An object of S3 class will not complain when such things happen, but an object of S4 class will not allow such mistakes.
- S4 classes are much more formal when we are creating objects, defining the class, etc.

S4 classes: example

The `setClass` function will create an S4 class, whilst `new` will create S4 objects:

```
setClass('employee',
  representation(name = 'character', salary = 'numeric', union = 'logical'))

joe <- new('employee', name = 'Joe', salary = 55000, union = TRUE)
joe
```

```
## An object of class "employee"
## Slot "name":
## [1] "Joe"
##
## Slot "salary":
## [1] 55000
##
## Slot "union":
## [1] TRUE
```

Printing S4 objects

S4 objects contain values in **slots** rather than **tags**.

Slots are accessed with an `@` sign or the slot function:

```
joe@salary
```

```
## [1] 55000
```

```
slot(joe, 'union')
```

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

We can update objects in the same way as a `list`:

```
joe@salary <- 65000
```

. . . but it won't allow us to add new elements:

```
joe$salary <- 55000
```

```
## Error in `$<-`(`*tmp*`, salary, value = 55000): no method for assigning subsets of this S4 class
```

The function `show` is the S4 equivalent of `print`.

Creating generic functions

The `setMethod` function creates new S4 generic functions:

```
setMethod('show', 'employee',
  function(object) {
  inorout <- ifelse(object@union, 'is', 'is not')
  cat(object@name, 'has a salary of', object@salary, 'and',
    inorout, 'in the union \n')
  }
)
joe
```

```
## Joe has a salary of 65000 and is in the union
```

S3 vs S4

- Online arguments rage about the superiority of S3 vs S4 (similar to `=` vs `<-`).
- S3 is convenient and simple to use, whilst S4 is very safe and makes it harder to make mistakes
- Google's R style guide (https://google.github.io/styleguide/Rguide.xml) recommends not using S4 classes
- However the founders of the R language say that S4 classes are needed to write 'clear and reliable software'.

Managing objects

- More on `ls()` and `ls.str()`
- Loading and saving objects
- Tracking objects in the workspace
- `exists()`

Managing objects

- It's very easy to accumulate a large number of objects in an R session.
- In this section we will look at a number of functions that enable you to keep track of objects, some of which we've met before:
  - `ls()` and `ls.str()` functions.
  - `save()` and `save.image()` functions.
  - `class()` and `mode()`.
  - `exists()`.

More on `ls()` and `ls.str()`.

- We met the `ls()` and `ls.str()` functions in lecture 6 when covering environments and scope.
- `ls()` lists the objects in the current workspace and `ls.str()` additionally gives their structure.
  - You can add the `pattern` argument to search for certain strings in the workspace

```
ls(pattern = 'jo')
```

```
## [1] "joe"
```

You can remove objects from the workspace with `rm`:

```r
rm(joe, x, y)
```

A quick way of removing everything from the workspace is:

```r
rm(list = ls())
```

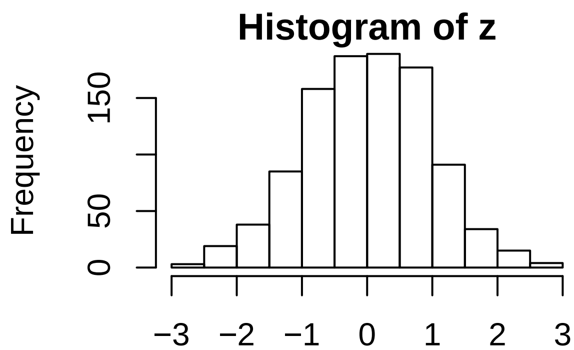
Remember the environment tab in Rstudio will also show you which objects are in your workspace.

Savings objects

The `save()` and `load()` functions will save/load objects to/from disk.

`save.image()` will save all objects in the workspace.

```r
z <- rnorm(1000)
hz <- hist(z)
```




```r
# save the histogram
save(hz, file = 'hzfile.RData')
ls()
```

```
## [1] "hz" "z"
```

```r
rm(hz)
```

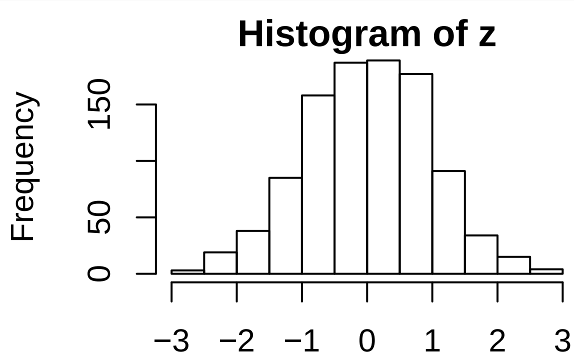
```r
ls()
```

```
## [1] "z"
```

```r
# load the histogram
load('hzfile.RData')
ls()
```

```
## [1] "hz" "z"
```

```r
plot(hz)
```



What are these objects?

- Often a call to `ls()` will yield objects that we didn't know were there and want to find out more about.
- Printing large objects is often not an option as it can take too long and will fill the display.
- Instead we can use:
  - `str()` to give the structure.
  - `mode()` and `class()` to determine the data type and class.
  - `unclass()` is useful to see the underlying structure of an object in case the `print` or `summary` methods obscure it.
  - `names()` and `attributes()` give details about the tags in a list.

Does it exist?

- When you are loading/saving/removing objects in the workspace you can often lose track of which objects are available to you.
- The function `exists()` will return a logical variable telling you whether that object is in the workspace or not.

Note: the object given to `exists` needs to be in inverted commas:

```
exists('hz')
```

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

```
exists(hz)
```

```
## Error in exists(hz): invalid first argument
```

- This allows you to give lists of objects to exist if required.

Lessons from this week

- R uses object-oriented programming. Everything is an object.
- Two types of class: S3 and S4. S3 is simple to use but S4 has some nice extra safety features.
- Make sure to keep track of the objects in your workspace.