Advanced Programming

Lecture 4

Nicholas Christian

BIOST 2094 Spring 2011

Outline

1. Classes and Methods
2. S3 Classes and Methods
3. S4 Classes and Methods
4. Attributes
5. Advanced Functions
6. Error Recovery

Classes and Methods

Everything in `R` is an OBJECT.

- A *class* is the definition of an object.
- A *method* is a function that performs specific calculations on objects of a specific class. A *generic function* is used to determine the class of its arguments and select the appropriate method. A generic function is a function with a collection of methods.
- See `?Classes` and `?Methods` for more information.

Example - Classes and Methods

- For example, consider the `ToothGrowth` dataset that is included with `R`. This dataset looks at the effect of vitamin C on tooth growth in guinea pigs. (This is an example of S3 classes and methods.)
  - `ToothGrowth` is an object, the class of `ToothGrowth` is a dataframe
  - `summary()` is a generic function that calls the method `summary.data.frame()` to summarize the contents of `ToothGrowth`
  - `lm()` returns a linear model object and `summary()` calls the method `summary.lm()` to summarize the results of the model fit

```
class(ToothGrowth)
summary(ToothGrowth)
summary.data.frame(ToothGrowth)

fit <- lm(len ~ dose + factor(supp), data=ToothGrowth)
class(fit)
summary(fit)
summary.lm(fit)
```

S3 and S4 Classes

- There are two types of classes in R,
  - S3 Classes - old style, quick and dirty, informal
  - S4 Classes - new style, rigorous and formal
- The S3 approach is very simple to implement. Just add a class name to an object. There are no formal requirements about the contents of the object, we just expect the object to have the right information.
- An S4 class gives a rigorous definition of an object. Any valid object of an S4 class will meet all the requirements specified in the definition.
- Despite the shortcomings of S3 classes they are widely used and unlikely to disappear.

Why Classes and Methods

- S4 classes reduce errors. When a function acts on a S4 class object it knows what type of information is in the object and that the information is valid.
- Methods simplify function calls and make computations more natural and clearer. Consider `summary()`, with methods we no longer need a separate function to summarize each type of object. Instead we can have one function that we call for summarizing a variety of object types.
- With S4 classes it is easier to organize and coordinate the work of multiple contributors for large complicated projects.
  - For example, The bioconductor package used for microarray data makes extensive use of S4 classes and methods.
  - See the biconductor website for more information, `http://www.bioconductor.org/`

Example - New Classes and Methods

- As an example we will create an S3 and S4 infant development class, `infant`. Objects of this class will contain information on the growth of an infant from birth to 36 months.
- The object will contain,
  - An identification number
  - Infant's gender
  - A dataframe of the infant's age (months), height (cm), and weight (kg)

S3 Class and Methods

- To create an S3 class all we need to do is set the class attribute of the object using `class()`.
- To create an S3 method write a function with the name *`generic.class`*, where *`generic`* is a generic function name and *`class`* is the corresponding class for the method.
- Examples of generic functions are `summary()`, `print()` and `plot()`. See `?UseMethod` for how to create new generic functions for S3 methods.

`class(x)`	Get or set the class attributes of `x`
`unclass(x)`	Remove class attributes of `x`
`methods(generic.function)`	All available S3 methods for a generic function
`methods(class="`*`class`*`")`	Get all S3 methods for a particular class
`is(object)`	Return `object`'s class and all super-classes
`is(object, class2)`	Tests if `object` is from `class2`
`str(object)`	Display the internal structure of an `object`

infant S3 Class and Methods

```
# Create a class object infant
infant <- function(ID, sex, age, ht, wt) {
     out <- list(ID=ID, sex=sex, data=data.frame(Age=age, HT.cm=ht, WT.kg=wt))
     class(out) <- "infant"
     invisible(out)
}

# Print method for infant class
print.infant <- function(object) {
     cat("ID  =", object$ID, "\nSex =", object$sex, "\n")
     print(object$data)
}

# Plot method for infant class
plot.infant <- function(object) {
     data <- object$data

     par(mfrow=c(1,2))
     plot(data$Age, data$HT.cm, type="o", pch=19, col="blue",
          xlab="Age (months)", ylab="Height (cm)", main="Height vs Age")
     plot(data$Age, data$WT.kg, type="o", pch=19, col="blue",
          xlab="Age (months)", ylab="Weight (kg)", main="Weight vs Age")
     mtext(paste("Subject ",object$ID,", ",toupper(object$sex),sep=""), side=3,
            outer=TRUE, line=-1.5, cex=1.5)
}
```

infant S3 Class and Methods

- The "data" is the median height and weight for boys and girls from CDC growth charts, http://www.cdc.gov/growthcharts/

```
# Infant "data"
age       = c(0, 3, 6, 12, 18, 24, 30, 36)
male.wt   = c( 3.53,  6.39,  8.16, 10.46, 11.80, 12.74, 13.56, 14.33)
female.wt = c( 3.40,  5.86,  7.45,  9.67, 11.09, 12.13, 13.04, 13.87)
male.ht   = c(49.99, 62.08, 67.86, 76.11, 82.41, 87.66, 92.13, 95.45)
female.ht = c(49.29, 60.46, 66.12, 74.40, 80.80, 86.20, 91.13, 94.43)

# Create infant objects
x <- infant(1, "male", age, male.ht, male.wt)
y <- infant(2, "female", age, female.ht, female.wt)
class(x); class(y)

# Print infant objects
x; y
# Plot infant objects
plot(x); plot(y)

# Possible to create invalid objects with S3 classes
z <- infant(3, "male", c("0 mon", "3 mon"), c(49.99,62.08), c(3.53,6.39))
```

Printing Function Results

- Recall the `odds.ratio()` function from lecture 2. This function outputted the results to the console and returned a list object of the results invisibly.
- The results were printed using the `cat()` function and were always printed regardless of whether or not the function call was assigned.
- If we need to call this function repeatedly, such as in a simulation, the printed results can get in the way.
- One possible approach for suppressing the printed output is to include an indicator argument; that will indicate whether or not the results should be printed. A drawback of this approach is that we have to include an extra argument and the printing is done differently compared to the rest of `R`.
- A better approach is to use S3 classes. Remember that unassigned function calls will print the value returned and that assigned function calls will not print anything. So rather then including print statements within the function body, assign a class name to the object returned and write a print method. This way the object will print how you want it to, but it will also print in the same way `R` prints all of its other objects.

Odds Ratio Function - Original

```
odds.ratio <- function(X, conf.level=.95) {
    # Verify Arguments
    stopifnot(!missing(X), is.matrix(X), dim(X)==c(2,2), X>0)

    # Print warning message if any expected cell count < 5
    exp.count <- (apply(X, 1, sum) %o% apply(X, 2, sum))/sum(X)
    if(any(exp.count < 5)) warning("Expected cell count < 5")

    # Calculate odds ratio and asymptotic confidence interval
    OR <- (X[1,1]*X[2,2])/(X[1,2]*X[2,1])
    logOR.SE <- sqrt(sum(1/X))

    alpha <- 1-conf.level
    CI.lower <- exp(log(OR) - qnorm(1-alpha/2)*logOR.SE)
    CI.upper <- exp(log(OR) + qnorm(1-alpha/2)*logOR.SE)

    # Format and return results
    cat("Odds Ratio = ", OR, "\n",
        conf.level*100, "% Confidence Interval = (",
        CI.lower, ", ", CI.upper, ")\n", sep="")

    out <- list(OR=OR, CI=c(CI.lower, CI.upper), conf.level=conf.level)
    invisible(out)
}
```

Odds Ratio Function - Indicator Argument

```
odds.ratio.extraArg <- function(X, conf.level=.95, quietly=FALSE) {
    # Verify Arguments
    stopifnot(!missing(X), is.matrix(X), dim(X)==c(2,2), X>0)

    # Print warning message if any expected cell count < 5
    exp.count <- (apply(X, 1, sum) %o% apply(X, 2, sum))/sum(X)
    if(any(exp.count < 5)) warning("Expected cell count < 5")

    # Calculate odds ratio and asymptotic confidence interval
    OR <- (X[1,1]*X[2,2])/(X[1,2]*X[2,1])
    logOR.SE <- sqrt(sum(1/X))

    alpha <- 1-conf.level
    CI.lower <- exp(log(OR) - qnorm(1-alpha/2)*logOR.SE)
    CI.upper <- exp(log(OR) + qnorm(1-alpha/2)*logOR.SE)

    # Format and return results, print if quietly=FALSE
    if(!quietly) {
       cat("Odds Ratio = ", OR, "\n",
           conf.level*100, "% Confidence Interval = (",
           CI.lower, ", ", CI.upper, ")\n", sep="")
    }
    out <- list(OR=OR, CI=c(CI.lower, CI.upper), conf.level=conf.level)
    invisible(out)
}
```

Odds Ratio Function - Class

```
odds.ratio.class <- function(X, conf.level=.95) {
    # Verify Arguments
    stopifnot(!missing(X), is.matrix(X), dim(X)==c(2,2), X>0)

    # Print warning message if any expected cell count < 5
    exp.count <- (apply(X, 1, sum) %o% apply(X, 2, sum))/sum(X)
    if(any(exp.count < 5)) warning("Expected cell count < 5")

    # Calculate odds ratio and asymptotic confidence interval
    OR <- (X[1,1]*X[2,2])/(X[1,2]*X[2,1])
    logOR.SE <- sqrt(sum(1/X))

    alpha <- 1-conf.level
    CI.lower <- exp(log(OR) - qnorm(1-alpha/2)*logOR.SE)
    CI.upper <- exp(log(OR) + qnorm(1-alpha/2)*logOR.SE)

    # Format and return results
    out <- list(OR=OR, CI=c(CI.lower, CI.upper), conf.level=conf.level)
    class(out) <- "OR"
    return(out)
}

print.OR <- function(object) {
     cat("Odds Ratio = ", object$OR, "\n", object$conf.level*100, "%
        Confidence Interval = (",object$CI[1],", ",object$CI[2],")\n", sep="")
}
```

Odds Ratio Function Output

```
X <- matrix(c(189, 104, 10845, 10933), nrow=2,
      dimnames=list("Treatment"=c("Placebo", "Aspirin"),
                    "Myocardial Infarction"=c("Yes", "No")))
# Original: results are always printed
> odds.ratio(X)
Odds Ratio = 1.832054
95% Confidence Interval = (1.440042, 2.330780)
> result <- odds.ratio(X)
Odds Ratio = 1.832054
95% Confidence Interval = (1.440042, 2.330780)

# Indicator: results printed only if quietly=FALSE
> odds.ratio.extraArg(X, quietly=FALSE)
Odds Ratio = 1.832054
95% Confidence Interval = (1.440042, 2.330780)
> result <- odds.ratio.extraArg(X, quietly=TRUE)

# OR Class: results printed if function call is not assigned
> odds.ratio.class(X)
Odds Ratio = 1.832054
95% Confidence Interval = (1.440042, 2.330780)
> result <- odds.ratio.class(X)
```

S4 Class and Methods

- S4 classes and methods are created using functions in the `methods` package, this package is loaded automatically when `R` is started.
- Information in S4 classes is organized into *slots*. Each slot is named and requires a specified class.
- Slots are one of the advantages of S4 classes. The class of the data in the slot must match the class corresponding to the slot. For example, it is not possible to have numeric data in a slot that is designated for character data.

`setClass()`	Create a new class
`setMethod()`	Create a new method
`setGeneric()`	Create a new generic function
`new()`	Generate a new object for a given class
`getClass()`	Get the class definition
`getMethod()`	Get the method definition
`getSlots()`	Get the name and class of each slot
`@`	Get or replace the contents of a slot
`validObject()`	Test the validity of an object

Create an S4 Class

- To create a new S4 class,

`setClass(Class, representation)`

`Center`	Name for the class
`representation`	Named list of the slots and the corresponding classes

- Use `new()` to generate a new object from a class

```
# Define an infant class
setClass("infant", representation(ID   = "numeric",
                                  sex  = "character",
                                  data = "data.frame"))
getClass("infant")
getSlots("infant")

# Create an object of class infant
x <- new("infant", data=data.frame(age, male.wt, male.ht), sex="male", ID=1)

x@data   # Use the @ sign to extract a specific slot

# The class of the data must match the class of the corresponding slot
# This is not the case with S3 classes, there we assume the class is right
y <- new("infant", data=data.frame(age, male.wt, male.ht), sex=1, ID="001")
```

Create S4 Methods

- To create an S4 method,

`setMethod(f, signature, definition)`

`f`	Name of the generic function
`signature`	Character string of the corresponding class
`definition`	Function definition

```
# Show method, similar to print for S3 classes
setMethod(f = "show", signature = "infant",
     definition = function(object) {
          cat("ID  =", object@ID, "\nSex =", object@sex, "\n")
          print(object@data)
     })

# Extract method
setMethod(f="[", signature="infant", definition=function(x,i,j) {x@data[i,j]})

# Replace method
setMethod(f = "[<-", signature = "infant",
          definition = function(x,i,j,value) {
            x@data[i,j] <- value
            validObject(x)
            return(x)
          })
```

Validity

- It is important that an object be valid. Slots help, but there maybe additional requirements. To test for these requirements supply a validity checking method to the validity argument of `setClass()`. This method should return `TRUE` if the object is valid and a description of the non-validity otherwise.

```
# Check the validity of an infant object
validity.infant <- function(object) {
     if(!all(sapply(object@data, is.numeric))) {
        return("data not numeric")
     } else return(TRUE)
}

setClass("infant",
         representation(ID   = "numeric",
                        sex  = "character",
                        data = "data.frame"),
         validity=validity.infant)

x <- new("infant", data=data.frame(age, male.wt, male.ht), sex="male", ID=1)
z <- new("infant", data=data.frame(c("0mon"),c(49.99),c(3.53)), sex="male", ID=1)
```

Inheritance

- Possible to create a new class that extends the first class. That is, create a new class that contains all of the information from the existing class, plus additional slots.
- Methods defined for the contained class can also be used for the new class.
- Validity requirements of the contained class also apply to the new class.
- Use the argument `contains` in `setClass()` to set a superclass. Set `contains` equal to a character string of the name of the class being extend.

Inheritance

- `infant` is a *superclass* of `infant.BirthOrder` or `infant.BirthOrder` is a *subclass* of `infant`. The terminology can be confusing since `infant` has fewer slots then `infant.BirthOrder`. The naming comes from the application of methods, any method that works for a superclass will work on a subclass, but not the other way around.
- We only need to create a new `show` method for `infant.BirthOrder`. We do not need to recreate the extract or replace methods.

```
setClass("infant.BirthOrder",
         representation(BirthOrder = "numeric"),
         contains="infant")

setMethod(f = "show", signature = "infant.BirthOrder",
     definition = function(object)
          cat("ID  =", object@ID, "\nSex =", object@sex,
              "\nBirth Order =", object@BirthOrder, "\n")
          print(object@data)
     )

x.more <- new("infant.BirthOrder", BirthOrder=1,
   data=data.frame(age, male.wt, male.ht), sex="male", ID=1)
```

Advantages of S4 Classes and Methods

- **Type** The class of the data in the slot must match the class corresponding to the slot. Thus, we can be certain that when we extract a slot the data is the appropriate type.
- **Validity** An object must meet specific requirements before it is created. As a result, we can be confident that an object is valid.
- **Inheritance** Objects can inherit properties from other objects that are already defined. Methods defined for a superclass can be applied to a subclass as well.
- **Encapsulation** After a class and the corresponding methods are defined, we no longer need to think about the internal code. We can just focus on using the classes and methods.
- **Planning** To be effective and take advantage of the inheritance property it is necessary to have an overall plan about what kinds of objects and methods are needed.

Attributes

- Properties or *attributes* of an object are given by `attributes()`.
- The function `attr()` is used to get or set the attributes of an object.
- The mode `mode()` and length `length()` are intrinsic attributes that are not returned by `attributes()`. The class attribute is not returned if the object has an implicit class like matrix.
- Attributes are useful for attaching information to an object. For example, it may be useful to save the function call but we don't want the call to be part of the object returned by the function. So we attach a "call" attribute to the object returned.

```
power <- function(x, n) {
            out <- x^n
            attr(out, "call") <- match.call()
            class(out) <- "new"
            return(out)
}
(y <- power(1:10, 2))
attributes(y)
attr(y, "call")
```

Advanced Functions

- Below are some functions that deal with unevaluated `R` expressions.

`eval()`	Evaluates an unevaluated `R` expression
`substitute()`	Returns an unevaluated expression, substituting any variables found in the current environment
`quote()`	Returns its unevaluated argument
`parse()`	Convert character strings to a parsed but unevaluated expression
`deparse()`	Turn unevaluated expressions into character strings

Example - `eval()` and `parse()`

- Sometimes it may be easier to write code using character string functions.
- In which case we need to use `parse()` to convert the character string to an unevaluated expression and then call `eval()` to evaluate this expression.
- For example, suppose I want to write a function that calculates the *n*th power of a matrix. The `paste()` function is an easy way to write out all of the matrix multiplications.

```
matrix.n <- function(A, n) {
     A.n <- paste(rep("A", n), collapse="%*%")
     eval(parse(text=A.n))
}
```

```
> x <- matrix(1:4, nrow=2)
> matrix.n(x, 5)
     [,1] [,2]
[1,] 1069 2337
[2,] 1558 3406
```

Example - deparse() and substitute()

- deparse() and substitute() are typically used to create character strings of argument values. These character strings can then be used to create more informative output.

```
getMean <- function(x) {
     var <- deparse(substitute(x))
     if(!is.numeric(x)) stop(paste(var, "is not numeric"))
     cat("The mean of", var, "is", mean(x), "\n")
}
```

```
> scores = c(90,78,88,93,85)
> getMean(scores)
The mean of scores is 86.8

> grades = c("A", "C", "B", "A", "B")
> getMean(grades)
Error in getMean(grades) : grades is not numeric
```

Error Recovery

- Sometimes when we are running a program, such as a simulation, we do not want the program to stop running when an error occurs, rather we would like the program to make a note of the error and continue running.

```
try(expr, silent = FALSE)
```

- The function `try()` evaluates an expression and traps any errors that occur during the evaluation.
- `try()` will return the value of the expression if it is evaluated without an error, but an invisible object of class "try-error" containing the error message if it fails. If `silent=TRUE` the error message is suppressed.
- See `tryCatch()` for a more general approach to catching and handling errors as well as warnings
- `suppressWarnings()` evaluates its expression and ignores all warnings messages

```
> log(-3)
[1] NaN
Warning message:
In log(-3) : NaNs produced
> suppressWarnings(log(-3))
[1] NaN
```

Example - Error Recovery

- Simulation study of the effect of very small uniformly distributed covariates on least square estimates.

```
# ---------------------------------------
sigma = 1      # Error standard deviation
 beta = c(1,2) # Coefficients
    n = 100    # Sample size
 nsim = 1000   # Number of simulations
# ---------------------------------------

beta.hat <- matrix(nrow=nsim, ncol=length(beta))
for(i in 1:nsim) {
     # Generate Data
     e <- rnorm(n, 0, sigma)
     X <- cbind(rep(1, n), runif(n, 0, 5.5E-8))
     Y <- X%*%beta + e

     # Estimate parameters
     t <- try(solve(t(X)%*%X)%*%t(X)%*%Y, silent=TRUE)
     if(is(t, "try-error")) next else beta.hat[i,] <- t
}
apply(beta.hat, 2, mean, na.rm=TRUE) # Avg estimates over successful iterations
sum(is.na(beta.hat[,1]))             # Number of failed iterations
```