R Programming

Object-Oriented Programming

(The S3 System)

Generic Language

- A good deal of the language we use to describe the world around us is ambiguous.
- This is because we tend to use a large number of generic terms whose meaning is made clear by the context they are used in.
- This ambiguous use of words is very useful because it enables us to get by with a much smaller vocabulary.

Examples

- “Fitting a model” can mean quite different things depending on the particular model and data being used.
- What we do when we “compute a data summary” depends very much on the kind of data we are summarising.
- The kind of plot produced when we “plot some data” is very dependent on the kind of data we are plotting.

Generic Functions

- Ambiguity is handled in R by introducing a programming facility based on the idea of *generic functions*.
- The use of generic functions means that the action that occurs when a function is applied to a set of arguments is dependent on the type of arguments the function is applied to.
- Because the action that occurs is determined by the objects passed as arguments, this kind of computer programming is known as *object-oriented programming*.

Types and Classes

- To really make use of object-oriented programming in R, we need a way of creating new types of object.
- In the object-oriented paradigm, these new object types are referred to as *classes*.
- Because the R object system was added some time after R was created, not all R objects are defined using the class system.
- However, even though they may not be defined using the class system, all R objects have an implied class that can be obtained with the `class` function.

Vectors

The basic vector types correspond to the classes with names `logical`, `numeric`, `character` and `complex`, while lists are associated with the `list` class.

```
> class(10)
[1] "numeric"

> class("hello")
[1] "character"

> class(list(1, 2, 1))
[1] "list"
```

More Complex Objects

Other, more complex, R objects also have an implied class. For example matrices are associated with the `matrix` class and higher-way arrays with the `array` class.

```
> class(matrix(1:4, nc = 2))
[1] "matrix"

> class(matrix(letters[1:4], nc = 2))
[1] "matrix"
```

Notice that although the underlying modes of these objects differ, they are both regarded as being of class `"matrix"`.

An Example

- We will develop software that manipulates numerical data associated locations in the *x*-*y* plane.
- Initially we will simply model the locations; later we will add the ability to associate numerical values with the locations.
- The actual locations will be stored in vectors that separately contain the *x* and *y* coordinates of the locations.

Creating the Locations

A simple way of storing the location vectors is in a list.

The following statements take two samples of size five from a normal distribution. To make them easier to read they are rounded to two decimal places.

```
> pts = list(x = round(rnorm(5), 2),
             y = round(rnorm(5), 2))

> pts
$x
[1] -0.35  0.72  0.25 -1.68 -0.31

$y
[1] 1.32 0.51 1.98 0.58 0.74
```

Adding a Class Attribute

To indicate the list, `pts` has a special structure we attach a class to it. This is done by assigning to the class of the object.

```
> class(pts) = "coords"

> pts
$x
[1] -0.35  0.72  0.25 -1.68 -0.31

$y
[1] 1.32 0.51 1.98 0.58 0.74

attr(,"class")
[1] "coords"
```

Constructor Functions

- Creating objects by simply attaching a class to a value is dangerous, because the value may not be appropriate.
- Because of this it is useful to wrap object creation inside a constructor function that can carry out some basic checks.
- In the case of `coords` object, we probably need to check the following:
  - The x and y values are numeric vectors.
  - The vectors contains no `NA`, `NaN` or `Inf` values.
  - The vectors have the same length.

The Constructor Function

```
> coords =
      function(x, y)
      {
          if (!is.numeric(x) || !is.numeric(y) ||
              !all(is.finite(x)) ||
              !all(is.finite(y)))
                  stop("invalid coordinates")
          if (length(x) != length(y))
              stop("coordinate lengths differ")
          pts = list(x = x, y = y)
          class(pts) = "coords"
          pts
      }
```

Using the Constructor

Once defined, the constructor function provides a simple way of creating objects from a class.

```
> pts = coords(x = round(rnorm(5), 2),
               y = round(rnorm(5), 2))

> pts
$x
[1] -0.29  0.56  0.55 -2.04 -1.48

$y
[1]  0.03 -1.34  1.01 -0.17 -1.05

attr(,"class")
[1] "coords"
```

Discarding Class Information

Sometimes it us useful to work with the information present in objects without the “baggage” of the class information.

A simple way to do this is to use the `unclass` function.

```
> unclass(pts)
$x
[1] -0.29  0.56  0.55 -2.04 -1.48

$y
[1]  0.03 -1.34  1.01 -0.17 -1.05
```

Accessor Functions

Although `coords` objects can be treated as a list it is better to define *accessor* functions that return the components.

This makes the software more modular and easier to modify.

```
> xcoords = function(obj) obj$x
> ycoords = function(obj) obj$y

> xcoords(pts)
[1] -0.29  0.56  0.55 -2.04 -1.48
> ycoords(pts)
[1]  0.03 -1.34  1.01 -0.17 -1.05
```

Generic Functions and Methods

- Class attributes are the basis for the simple “S3” object-oriented mechanism.
- The mechanism uses a special type of function called a *generic function*.
- A generic function acts as a kind of switch that selects a particular function or *method* to invoked.
- The particular method selected depends on the class of the first argument.

Generic Functions and Methods

- It will be useful to be able to print the values of coords objects in a more useful form than when they are printed as a list.
- To do that, we simply need to define a *method* for printing.
- The method is a function that has a name made by pasting together the method name and the class name, separated by a dot (“`.`”).
- The `print` function is generic and a “print method” can be defined for coords objects as follows.

  (The `print` function is invoked implicitly when the result of an expression is displayed.)

A Print Method

```
> print.coords =
      function(obj) {
          print(paste("(",
                      format(xcoords(obj)),
                      ", ",
                      format(ycoords(obj)),
                      ")", sep = ""),
                quote = FALSE)
      }

> pts
[1] (-0.29,  0.03) ( 0.56, -1.34)
[3] ( 0.55,  1.01) (-2.04, -0.17)
[5] (-1.48, -1.05)
```

Other Methods

- Methods can be defined for any functions that are generic.
- The `length` function is another example of a generic function.

```
> length.coords =
      function(obj) length(xcoords(obj))

> length(pts)
[1] 5
```

- A function's manual entry will indicate whether or not it is generic.

Creating Generic Functions

- Methods can only be defined for functions that are *generic*.
- New generic functions are easy to create.
- Generic functions have a very special form.

```
> bbox =
      function(obj)
      UseMethod("bbox")
```

Creating a `bbox` Method

```
> bbox.coords =
      function(obj) {
          bb = rbind(range(xcoords(obj)),
                     range(ycoords(obj)))
          dimnames(bb) = list(c("x:", "y:"),
                              c("min", "max"))
          bb
      }

> bbox(pts)
     min  max
x: -2.04 0.56
y: -1.34 1.01
```

Adding Values to the Coordinates

- We can now set about creating objects that contain both locations and values.
- We will do this by creating a new class of object called `vcoords` (for *values with coordinates*).
- This new kind of object will be tagged with both the new class name and the old one, so that we can continue to use the methods we have already defined.

The Constructor Function

```
> vcoords =
      function(x, y, v)
      {
          if (!is.numeric(x) || !is.numeric(y) ||
              !is.numeric(v) ||
              !all(is.finite(x)) ||
              !all(is.finite(y)))
                  stop("invalid coordinates")
          if(length(x) != length(y) ||
             length(x) != length(v))
              stop("argument lengths differ")
          pts = list(x = x, y = y, v = v)
          class(pts) = c("vcoords", "coords")
          pts
      }
```

Using the Constructor

```
> pts = vcoords(x = round(rnorm(5), 2),
                y = round(rnorm(5), 2),
                v = round(runif(5, 0, 100)))
```

It will also be useful to have a new accessor function.

```
> values = function(obj) obj$v
> values(pts)
[1] 54 81 11 40 66
```

Unfortunately, the new objects do not display as expected.

```
> pts
[1] (0.89, -1.09) (0.02, -0.31) (0.37, -1.07)
[4] (1.13, -0.74) (0.49, -1.57)
```

Defining a Print Method

Clearly we need an appropriate method for `vcoords` objects, we can't just use the `coords` one.

```
> print.vcoords =
      function(obj) {
          print(paste("(",
                      format(xcoords(obj)),
                      ", ",
                      format(ycoords(obj)),
                      "; ",
                      format(values(obj)),
                      ")", sep = ""),
                quote = FALSE)
      }
```

Results

- We can now get a sensible printed result for `vcoords` objects.

```
> pts
[1] (0.89, -1.09; 54) (0.02, -0.31; 81)
[3] (0.37, -1.07; 11) (1.13, -0.74; 40)
[5] (0.49, -1.57; 66)
```

- But notice that the `bbox` method for `vcoords` produces a sensible result and does not need to be redefined.

```
> bbox(pts)
     min   max
x:  0.02  1.13
y: -1.57 -0.31
```

Mathematical Transformations

- `vcoords` objects contain a numeric slot that can be changed by mathematical transformations.
- We may wish to:
  - apply a mathematical function to the values,
  - negate the values,
  - add, subtract, multiply or divide corresponding values in two objects,
  - compare values in two objects,
  - etc.
- Defining appropriate methods will allow us to do this.

Mathematical Functions

Here is how we can define a cos method.

```
> cos.vcoords =
      function(x)
      vcoords(xcoords(x),
              ycoords(x),
              cos(values(x)))

> cos(pts)
[1] (0.89, -1.09; -0.829309833)
[2] (0.02, -0.31;  0.776685982)
[3] (0.37, -1.07;  0.004425698)
[4] (1.13, -0.74; -0.666938062)
[5] (0.49, -1.57; -0.999647456)
```

Mathematical Functions

A sin method is very similar.

```
> sin.vcoords =
      function(x)
      vcoords(xcoords(x),
              ycoords(x),
              sin(values(x)))

> sin(pts)
[1] (0.89, -1.09; -0.55878905)
[2] (0.02, -0.31; -0.62988799)
[3] (0.37, -1.07; -0.99999021)
[4] (1.13, -0.74;  0.74511316)
[5] (0.49, -1.57; -0.02655115)
```

Group Methods - `Math`

In fact most of R’s mathematical functions would require an almost identical definition.

Fortunately, there is a short-hand way of defining all the methods with one function definition.

We define a “group method” for all the standard mathematical transformations.

```
> Math.vcoords =
      function(x)
      vcoords(xcoords(x),
              ycoords(x),
              get(.Generic)(values(x)))
```

The expression `get(.Generic)` gets the function with the name that `Math.vcoords` was invoked under.

Results

```
> sqrt(pts)
[1] (0.89, -1.09; 7.348469)
[2] (0.02, -0.31; 9.000000)
[3] (0.37, -1.07; 3.316625)
[4] (1.13, -0.74; 6.324555)
[5] (0.49, -1.57; 8.124038)

> log(pts)
[1] (0.89, -1.09; 3.988984)
[2] (0.02, -0.31; 4.394449)
[3] (0.37, -1.07; 2.397895)
[4] (1.13, -0.74; 3.688879)
[5] (0.49, -1.57; 4.189655)
```

Group Methods – `Ops`

- In addition to mathematical transformations, it is useful to define arithmetic methods for `vcoords` objects.
- Defining a method for the `Ops` generic makes it possible to simultaneously define methods for all the following binary operations.
  - `"+"`, `"-"`, `"*"`, `"/"`, `"^"`, `"%%"`, `"%/%"`,
  - `"&"`, `"|"`, `"!"`,
  - `"=="`, `"!="`, `"<"`, `"<="`, `">="`, `">"`.
- In order for these methods to work correctly, we need to ensure that locations of the vcoords values being operated on are identical.

A Simple `Ops` Method for `vcoords` Objects

```
> sameloc =
      function(e1, e2)
      (length(values(e1)) == length(values(e2))
       || all(xcoords(e1) == xcoords(e2))
       || all(ycoords(e1) == ycoords(e2)))

> Ops.vcoords =
      function(e1, e2) {
          if (!sameloc(e1, e2))
              stop("different locations")
          else vcoords(xcoords(e1),
                       ycoords(e2),
                       get(.Generic)(values(e1),
                                     values(e2)))
      }
```

Results

```
> pts + pts
[1] (0.89, -1.09; 108) (0.02, -0.31; 162)
[3] (0.37, -1.07;  22) (1.13, -0.74;  80)
[5] (0.49, -1.57; 132)

> pts * pts
[1] (0.89, -1.09; 2916) (0.02, -0.31; 6561)
[3] (0.37, -1.07;  121) (1.13, -0.74; 1600)
[5] (0.49, -1.57; 4356)
```

Complexities

- Unfortunately, things are more complex than they seem on the surface.
- Comparisons (`==`, `!=`, `<`, `<=`, etc.) return logical values so creating `vcoords` result is not appropriate in those cases.
- It makes sense to be able to carry out arithmetic on a mix of `vcoords` objects and numeric values.
- This means that either or both arguments to binary operators can be a `vcoords` object. More type checking is needed.

Type and Sanity Checks

- It is possible to check whether an object `obj` is of the class `vcoords` with the expression

  ```
  inherits(obj, "vcoords")
  ```

- To keep things simple, we'll assume that if one argument to an `Ops` function is of class `vcoords` and the other is numeric, the numeric argument must be shorter than the `vcoords` object.

- The code for the full `Ops` method is rather too long to fit on a single slide. You'll have to look at it on.

The Ops Group Method

```
Ops.vcoords =
    function(e1, e2) {
        if (missing(e1)) {

            code for unary operations

        }
        else {

            code for binary operations

        }

        return the computed value
}
```

Code for Unary Operations

For unary operations, we get the *x* and *y* coordinates from the object and apply the generic function to the *values* from the object.

```
x = xcoords(e1)
y = ycoords(e1)
v = get(.Generic)(values(e1))
```

Code for Binary Operations

For binary operations we need to know which of the two arguments is a vcoords object.

```
is.vc1 = inherits(e1, "vcoords")
is.vc2 = inherits(e2, "vcoords")
if (is.vc1 && is.vc2) {
    both objects are vcoords
}
else if (is.vc1) {
    just the first object is vcoords
}
else if (is.vc2) {
    just the second object is vcoords
}
```

Both Objects Vcoords

If both objects are `vcoords` we have to check that they have the same locations (and hence have the same lengths).

If they do we extract the *x* and *y* locations (from the first object) and transform the values with the generic function.

```
if (!sameloc(e1, e1))
    stop("different locations")
x = xcoords(e1)
y = ycoords(e1)
v = get(.Generic)(values(e1), values(e2))
```

One Object is Vcoords

If just the first object is `vcoords` we use its locations and combine its values with the second argument using the generic function.

We need to check that the second value is no longer than the first.

```
if (length(e1) < length(e2))
    stop("size mismatch")
x = xcoords(e1)
y = ycoords(e1)
v = get(.Generic)(values(e1), e2)
```

The other case is similar.

Returning a Value

The preceding code computes all the components that are needed to create the return value (`x`, `y` and `v`).

The type of value to be returned depends on the operation being carried out. Arithmetic produces a `vcoords` object and comparison produces a logical result.

The choice can be made using an `if` statement or using a `switch`.

```
switch(.Generic,
       "=="=, "!="=, "<"=, "<="=, ">"=, ">="= v,
       vcoords(x, y, v))
```

Returning a Value (“If” Alternative)

Use either:

```
if (.Generic == "==" || .Generic == "!=" ||
    .Generic == "<"  || .Generic == "<=" ||
    .Generic == ">"  || .Generic == ">=")
    v
else
    vcoords(x, y, v))
```

or:

```
if (!is.na(match(.Generic, c("=", "!=",
                             "<", "<=",
                             ">", ">="))))
    v
else
    vcoords(x, y, v))
```

Results

```
> pts
[1] (0.89, -1.09; 54) (0.02, -0.31; 81)
[3] (0.37, -1.07; 11) (1.13, -0.74; 40)
[5] (0.49, -1.57; 66)

> 2 * pts
[1] (0.89, 0.89; 108) (0.02, 0.02; 162)
[3] (0.37, 0.37;  22) (1.13, 1.13;  80)
[5] (0.49, 0.49; 132)

> pts > 50
[1]  TRUE  TRUE FALSE FALSE  TRUE
```

Subsetting

- Clearly we may wish to have access to subsetting methods when dealing with `vcoords` objects.
- Expressions like

  ```
  pts[xcoords(pts) < 0 & ycoords(pts) < 0]
  ```

  need to be defined.
- This can be handled by defining methods for `[`.

A Subsetting Method

```
> `[.vcoords` = function(x, i)
      vcoords(xcoords(x)[i], ycoords(x)[i],
              values(x)[i])

> pts
[1] (0.89, -1.09; 54) (0.02, -0.31; 81)
[3] (0.37, -1.07; 11) (1.13, -0.74; 40)
[5] (0.49, -1.57; 66)
> pts[1:3]
[1] (0.89, -1.09; 54) (0.02, -0.31; 81)
[3] (0.37, -1.07; 11)
> pts[pts > 50]
[1] (0.89, -1.09; 54) (0.02, -0.31; 81)
[3] (0.49, -1.57; 66)
```

Assigning to Subsets

- It is common to want to alter the values stored in some subset of an object.
- The syntax for this kind of operation is as follows.

```
var[subset-specification] = values
```

- This expression above is formally equivalent to the following.

```
var = `[<-`(var, subset-specification, values)
```

- The function `[<-` is generic and it is possible to write methods for it.

Defining a Subset Assignment Method

```
> `[<-.vcoords` =
      function(x, i, values) {
          if (!inherits(values, "vcoords"))
              stop("invalid right-hand side")
          if (length(values) != length(i))
              stop("invalid replacement length")
          xx = xcoords(x)
          xy = ycoords(x)
          xv = values(x)
          xx[i] = xcoords(values)
          xy[i] = ycoords(values)
          xv[i] = values(values)
          vcoords(xx, xy, xv)
      }
```

Example: Subset Replacement

```
> pts
[1] (0.89, -1.09; 54) (0.02, -0.31; 81)
[3] (0.37, -1.07; 11) (1.13, -0.74; 40)
[5] (0.49, -1.57; 66)

> pts[1:2] = pts[3:4]

> pts
[1] (0.37, -1.07; 11) (1.13, -0.74; 40)
[3] (0.37, -1.07; 11) (1.13, -0.74; 40)
[5] (0.49, -1.57; 66)
```

General Replacement

- In general, it is possible to define a meaning for any expression of the form

  ```
  fun(var, args) = values
  ```

- This is taken to mean

  ```
  var = `fun<-`(var, args, values)
  ```

  so the appropriate effect can be obtained by providing the right definition of `` `fun<-` ``.

- The evaluator in R also ensures that nested calls like

  ```
  values(pts)[1:3] = NA
  ```

  will work correctly.

Example: General Replacement

```
> `values<-` =
      function(x, values) {
        if (length(values) != length(x))
          stop("invalid replacement length")
        vcoords(xcoords(x), ycoords(x), values)
      }

> values(pts)[1:3] = NA
> pts
[1] (0.37, -1.07; NA) (1.13, -0.74; NA)
[3] (0.37, -1.07; NA) (1.13, -0.74; 40)
[5] (0.49, -1.57; 66)
```

Problems with the S3 Object System

- The S3 object system provides a range of object-oriented facilities, but does so in a very “loose” fashion.
- The following statements are permissible in R.

```
> model = 1:10; class(model) = "lm"

> class(x) = sample(class(x))
```

- Because this kind of manipulation is allowed, the S3 object system is not completely trustworthy.
- There are other more technical issues with the S3 system.