R Programming

Object-Oriented Programming

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 a 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 which occurs when a function is applied to a set of arguments is dependent on the type of arguments the function is being applied to.
- Because the action which occurs is determined by the objects passed as arguments, this kind of computer programming is known as *object-oriented programming*.

Types and Classes

- To 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 which 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 which 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 which 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.

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

> pts
$x
[1]  0.69 -0.34  1.59  0.12 -0.40

$y
[1] -1.20  0.68 -0.23  0.61  0.57
```

Adding a Class

To indicate the list has a special structure we attach a class to it.

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

> pts
$x
[1]  0.69 -0.34  1.59  0.12 -0.40

$y
[1] -1.20  0.68 -0.23  0.61  0.57

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 which 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

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

> pts
$x
[1]  0.22 -0.03  1.27  0.37 -0.37

$y
[1]  0.83 -1.02  1.65  1.32  0.40

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]  0.22 -0.03  1.27  0.37 -0.37

$y
[1]  0.83 -1.02  1.65  1.32  0.40
```

Accessor Functions

- Although `coords` objects can be treated as a list it is better to define *accessor* functions which 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]  0.22 -0.03  1.27  0.37 -0.37
> ycoords(pts)
[1]  0.83 -1.02  1.65  1.32  0.40
```

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 cooords 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.

A Print Method

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

> pts
[1] ( 0.22,  0.83) (-0.03, -1.02)
[3] ( 1.27,  1.65) ( 0.37,  1.32)
[5] (-0.37,  0.40)
```

Other Methods

- We can define methods for any functions which are defined as generic.

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

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

Creating Generic Functions

- Methods can only be defined for functions which are *generic*.
- New generic functions are easy to create.

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

Creating a `bbox` Method

```
> bbox.coords =
      function(obj)
      matrix(c(range(xcoords(obj)),
               range(ycoords(obj))),
             nc = 2,
             dimnames = list(
               c("min", "max"),
               c("x:", "y:")))

> bbox(pts)
        x:    y:
min -0.37 -1.02
max  1.27  1.65
```

Adding Values to the Coordinates

- We can now set about creating an object which contains both locations and values.
- We will do this by creating a new kind of `vcoords` object.
- This 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 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)))

> values = function(obj) obj$v

> values(pts)
[1] 18 79 64 56  4

> pts
[1] (-1.70, -0.08) (-0.11, -0.42)
[3] (-0.10,  0.36) ( 1.28,  0.93)
[5] (-0.06, -0.86)
```

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] (-1.70, -0.08; 18) (-0.11, -0.42; 79)
[3] (-0.10,  0.36; 64) ( 1.28,  0.93; 56)
[5] (-0.06, -0.86;  4)
```

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

```
> bbox(pts)
       x:    y:
min -1.70 -0.86
max  1.28  0.93
```

Mathematical Transformations

- `vcoords` objects contain a numeric slot which 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] (-1.70, -0.08;  0.6603167)
[2] (-0.11, -0.42; -0.8959709)
[3] (-0.10,  0.36;  0.3918572)
[4] ( 1.28,  0.93;  0.8532201)
[5] (-0.06, -0.86; -0.6536436)
```

Mathematical Functions

A sin method is very similar.

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

> sin(pts)
[1] (-1.70, -0.08; -0.7509872)
[2] (-0.11, -0.42; -0.4441127)
[3] (-0.10,  0.36;  0.9200260)
[4] ( 1.28,  0.93; -0.5215510)
[5] (-0.06, -0.86; -0.7568025)
```

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.

```
> 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] (-1.70, -0.08; 4.242641)
[2] (-0.11, -0.42; 8.888194)
[3] (-0.10,  0.36; 8.000000)
[4] ( 1.28,  0.93; 7.483315)
[5] (-0.06, -0.86; 2.000000)

> log(pts)
[1] (-1.70, -0.08; 2.890372)
[2] (-0.11, -0.42; 4.369448)
[3] (-0.10,  0.36; 4.158883)
[4] ( 1.28,  0.93; 4.025352)
[5] (-0.06, -0.86; 1.386294)
```

Group Methods – `Ops`

- 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.

`Ops` Methods 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] (-1.70, -0.08;  36) (-0.11, -0.42; 158)
[3] (-0.10,  0.36; 128) ( 1.28,  0.93; 112)
[5] (-0.06, -0.86;   8)

> pts * pts
[1] (-1.70, -0.08;  324) (-0.11, -0.42; 6241)
[3] (-0.10,  0.36; 4096) ( 1.28,  0.93; 3136)
[5] (-0.06, -0.86;   16)
```

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.
- Only one of the arguments to binary operators need to be a `vcoords` object. More type checking is needed.

Code Sketch

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

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

- If one argument to an `Ops` function is of class `vcoords` and the other is numeric, the numeric argument should be shorter than the `vcoords` object.

Results

```
> pts
[1] (-1.70, -0.08; 18) (-0.11, -0.42; 79)
[3] (-0.10,  0.36; 64) ( 1.28,  0.93; 56)
[5] (-0.06, -0.86;  4)

> 2 * pts
[1] (-1.70, -1.70;  36) (-0.11, -0.11; 158)
[3] (-0.10, -0.10; 128) ( 1.28,  1.28; 112)
[5] (-0.06, -0.06;   8)

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

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] (-1.70, -0.08; 18) (-0.11, -0.42; 79)
[3] (-0.10,  0.36; 64) ( 1.28,  0.93; 56)
[5] (-0.06, -0.86;  4)
> pts[1:3]
[1] (-1.70, -0.08; 18) (-0.11, -0.42; 79)
[3] (-0.10,  0.36; 64)
> pts[pts > 50]
[1] (-0.11, -0.42; 79) (-0.10,  0.36; 64)
[3] ( 1.28,  0.93; 56)
```

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] (-1.70, -0.08; 18) (-0.11, -0.42; 79)
[3] (-0.10,  0.36; 64) ( 1.28,  0.93; 56)
[5] (-0.06, -0.86;  4)

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

> pts
[1] (-0.10,  0.36; 64) ( 1.28,  0.93; 56)
[3] (-0.10,  0.36; 64) ( 1.28,  0.93; 56)
[5] (-0.06, -0.86;  4)
```

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] (-0.10,  0.36; NA) ( 1.28,  0.93; NA)
[3] (-0.10,  0.36; NA) ( 1.28,  0.93; 56)
[5] (-0.06, -0.86;  4)
```

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.

The S4 Object System

- Because of problems with the S4 object system, John Chambers has moved to a more formally-based object system.
- This "S4" system is quite similar to the Common Lisp CLOS object system or the object system in the Dylan language.
- The system is still "in development," but already offers significant advantages over the S3 system.
- One of the greatest advantages is the use of a formal system of inheritance.

Formal Classes

- In the formal class system, objects belong to formally defined classes. A class consists of a number of named *slots*, each with a specific type or class.
- A class which represents coordinates in the plane might have two slots, named `x` and `y`, which contain numeric vectors of the same length.
- Classes are be *declared* with a `setClass` statement. We could create a class to represent coordinates as follows.

```
> setClass("coords",
           representation(x = "numeric",
                          y = "numeric"))
[1] "coords"
```

Creating Objects

- Once a class has been created, objects from that class can be created with a call to the `new` function.

```
> pts = new("coords",
            x = rnorm(5), y = rnorm(5))
```

- The first argument to the function is the class name; the other arguments provide values for the slots in the object.

Constructor Functions

Generally, it is not a good idea to use `new` in this naked fashion. Instead it is better to embed object creation in a *constructor* function. This makes it possible to carry out some checks of slot validity.

```
> coords =
      function(x, y) {
          if (length(x) != length(y))
              stop("equal length x and y required")
          if (!is.numeric(x) || !is.numeric(y))
              stop("numeric x and y required")
          new("coords", x = as.vector(x),
                        y = as.vector(y))
      }
```

Constructing and Printing Objects

Objects from the `coords` class, known as *instances* of the class, can now be created with a call to the constructor function.

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

This kind of object can be printed just like any other R value.

```
> pts
An object of class "coords"
Slot "x":
[1]  1.31  1.15 -1.15 -0.46 -0.97

Slot "y":
[1]  0.64 -0.31  0.87  0.51 -1.34
```

Accessing an Object's Slots

- The values in the slots within an object can be accessed with the *slot access operator* `@`.
- The slots are accessed by name.

```
> pts@x
[1]  1.31  1.15 -1.15 -0.46 -0.97

> pts@y
[1]  0.64 -0.31  0.87  0.51 -1.34
```

Accessor Functions

- The code fragments `pts@x` and `pts@y` reveal a little too much of the internal structure of the `coords` class.
- Rather than getting the values directly in this way, it is better to use *accessor functions* which provide indirect access.
- That way the internal structure of the class can be changed more easily.

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

Generic Functions and Methods

- Formal classes are the basis for a clean object-oriented mechanism in R.
- 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 a number of nominated arguments.
- The types of the nominated arguments define the *signature* of the method.

The "show" Generic Function

- The display (i.e. printing) of objects is handled by the show generic function.
- We can see that show is a special kind of function if we print it.

```
> show
standardGeneric for "show"
    defined from package "methods"

function (object)
standardGeneric("show")
<environment: 0x8e34728>
Methods may be defined for arguments: object
Use  showMethods("show")  for currently available
(This generic function excludes non-simple inherit
```

Defining a "`show`" Method

Methods for show have a single argument called `object`. An appropriate `show` method for `coords` objects can be defined as follows. Here is how we could create a display method for the `coords` class defined above.

```
> setMethod(show, signature(object = "coords"),
            function(object)
            print(data.frame(x = xcoords(object),
                             y = ycoords(object))))
[1] "show"
attr(,"package")
[1] "methods"
```

Notice that the slots are accessed using the accessor functions rather than directly.

Using a Method

The *show* method will be used whenever an (implicit or explicit) attempt is made to print an object.

```
> pts
      x     y
1  1.31  0.64
2  1.15 -0.31
3 -1.15  0.87
4 -0.46  0.51
5 -0.97 -1.34
```

Defining New Generic Functions

If a function is not generic, it possible to create a new generic using the `setGeneric` function.

```
> setGeneric("display",
             function(obj)
             standardGeneric("display"))
[1] "display"
```

Writing a "display" Method

Once a function is defined as generic, new methods can be written for it. In the case of the `coords` class, we may choose to display the coordinates as pairs of values. It is easy to implement a method which will do this.

```
> setMethod("display", signature(obj = "coords"),
            function(obj)
            print(paste("(",
                        format(xcoords(obj)),
                        ", ",
                        format(ycoords(obj)),
                        ")", sep = ""),
                  quote = FALSE))
[1] "display"
```

Using the “`display`” Method

A call to the generic function `display` will be dispatched to the method just defined when the argument is of class `coords`.

```
> display(pts)
[1] ( 1.31,  0.64) ( 1.15, -0.31)
[3] (-1.15,  0.87) (-0.46,  0.51)
[5] (-0.97, -1.34)
```

Bounding Boxes

- One thing we might be interested in having for objects like those in the `coords` class, is a `bbox` method which will compute the two dimensional bounding box for the coordinates.

- We'll make the function generic so that methods can be defined for other classes too.

```
> setGeneric("bbox",
             function(obj)
             standardGeneric("bbox"))
[1] "bbox"
```

Implementing a Bounding Box Method

We can implement a bounding box method for the `coords` class as follows.

```
> setMethod("bbox", signature(obj = "coords"),
            function(obj)
            matrix(c(range(xcoords(obj)),
                     range(ycoords(obj))),
                   nc = 2,
                   dimnames = list(
                     c("min", "max"),
                     c("x:", "y:"))))
[1] "bbox"
```

Example: Bounding Box

```
> pts
      x     y
1  1.31  0.64
2  1.15 -0.31
3 -1.15  0.87
4 -0.46  0.51
5 -0.97 -1.34

> bbox(pts)
       x:    y:
min -1.15 -1.34
max  1.31  0.87
```

Inheritance

- The `coords` class provides a way to represent a set of spatial locations.
- Now suppose that we want a class which provides a numerical value to go along with each spatial location.
- We could define an entirely new class to do this, but it is better to simply add the value to our existing `coords` class.
- The new class will then *inherit* the spatial properties of the `coords` class.

Inheritance: The Class Declaration

Here is a declaration of the new class.

```
> setClass("vcoords",
           representation(value = "numeric"),
           contains = "coords")
[1] "vcoords"
```

This says that a `vcoords` object contains a numeric `value` slot and *inherits* the slots from the `coords` class.

Inheritance: A Constructor Function

```
> vcoords =
      function(x, y, value)
      {
          if (!is.numeric(x) ||
              !is.numeric(y) ||
              !is.numeric(value) ||
              length(x) != length(value) ||
              length(y) != length(value))
                  stop("invalid arguments")
          new("vcoords", x = x, y = y,
              value = value)
      }

> values = function(obj) obj@value
```

Example: A `vcoords` Object

Defining a `vcoords` object is simple

```
> vpts = vcoords(xcoords(pts), ycoords(pts),
                 round(100 * runif(5)))
```

but printing it gives an unexpected result.

```
> vpts
      x     y
1  1.31  0.64
2  1.15 -0.31
3 -1.15  0.87
4 -0.46  0.51
5 -0.97 -1.34
```

Inherited Methods

- The printing result occurs because the `vcoords` class doesn't just inherit the slots of the `coords` class. It also inherits its methods.
- A `vcoords` is also a `coords` object. When a search is is made for an appropriate `print` method, and no `vcoords` method is found, the `coords` method is used.
- If we want a method for printing `vcoords` objects, we have to define one.

A Print Method for vcoords Objects

```
> setMethod(show, signature(object = "vcoords"),
            function(object)
            print(data.frame(
                      x = xcoords(object),
                      y = ycoords(object),
                      value = values(object))))
[1] "show"
attr(,"package")
[1] "methods"
```

Printing `vcoords` Objects

```
> vpts
      x     y value
1  1.31  0.64     7
2  1.15 -0.31    79
3 -1.15  0.87    19
4 -0.46  0.51    98
5 -0.97 -1.34    79
```

Mathematical Transformations

- The `vcoords` class contains a numeric slot which 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 an cos method.

```
> setMethod("cos", signature(x = "vcoords"),
            function(x)
            vcoords(xcoords(x),
                    ycoords(x),
                    cos(values(x))))
[1] "cos"
> cos(vpts)
      x     y      value
1  1.31  0.64  0.7539023
2  1.15 -0.31 -0.8959709
3 -1.15  0.87  0.9887046
4 -0.46  0.51 -0.8192882
5 -0.97 -1.34 -0.8959709
```

Mathematical Functions

A sin method is very similar.

```
> setMethod("sin", signature(x = "vcoords"),
            function(x)
            vcoords(xcoords(x),
                    ycoords(x),
                    sin(values(x))))
[1] "sin"
> sin(vpts)
      x     y      value
1  1.31  0.64  0.6569866
2  1.15 -0.31 -0.4441127
3 -1.15  0.87  0.1498772
4 -0.46  0.51 -0.5733819
5 -0.97 -1.34 -0.4441127
```

Group Methods

In fact most of R's mathematical functions would require an almost identical definition. There is actually a short-hand way of defining all the methods with one function definition.

```
> setMethod("Math", signature(x = "vcoords"),
            function(x)
            vcoords(xcoords(x),
                    ycoords(x),
                    callGeneric(values(x))))
[1] "Math"
```

This provides definitions for all the common mathematical functions.

Group Methods

```
> sqrt(vpts)
      x     y    value
1  1.31  0.64 2.645751
2  1.15 -0.31 8.888194
3 -1.15  0.87 4.358899
4 -0.46  0.51 9.899495
5 -0.97 -1.34 8.888194

> tan(vpts)
      x     y     value
1  1.31  0.64 0.8714480
2  1.15 -0.31 0.4956775
3 -1.15  0.87 0.1515895
4 -0.46  0.51 0.6998537
5 -0.97 -1.34 0.4956775
```

Functions Handled by the Math Group

The following functions are handled by the `Math` group.

> `abs`, `sign`, `exp`, `sqrt`, `log`, `log10`, `log2`,
> `cos`, `sin`, `tan`, `acos`, `asin`, `atan`,
> `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`,
> `ceiling`, `floor`, `trunc`,
> `gamma`, `lgamma`, `digamma`, `trigamma`
> `cumprod`, `cumsum`, `cummin`, `cummin`.

There is also a `Math2` group (with a second argument called `digits`) which contains the following functions.

> `round`, `signif`.

Binary Operations

- There are many binary operations R.
- Examples are:
  - The arithmetic operators:

    `+`, `-`, `*`, `^`, `%%`, `%/%`, `/`
  - The comparison operators:

    `==`, `>`, `<`, `!=`, `<=`, `>=`
- These operators are all *generic*, and methods can be defined for them.
- The operators belong to the groups `Arith` and `Compare`, which both belong to the larger group `Ops`.

Binary Operators as Functions

- Any binary operator can be thought of as a function of two variables.
- The function has the form

```
function(e1, e2) {

. . .

}
```

- A call of the form `x + y` can be thought of as a call to a function like the one above.

Compatibility of `vcoords` Objects

- Arithmetic on `vcoords` objects only makes sense if the objects are defined at identical locations.
- Here is function which will check whether two `vccords` objects are defined at the same locations.

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

Defining Methods for Arithmetic Operators

Here is a group method which will define all the necessary binary operators for arithmetic on `vcoords` objects.

```
> setMethod("Arith", signature(e1 = "vcoords",
                               e2 = "vcoords"),
      function(e1, e2)
      {
          if (!sameloc(e1, e2))
              stop("identical locations required")
          vcoords(xcoords(e1),
                  ycoords(e1),
                  callGeneric(values(e1),
                              values(e2)))
      })
[1] "Arith"
```

Example: Adding `vcoords`

```
> vpts
      x     y value
1  1.31  0.64     7
2  1.15 -0.31    79
3 -1.15  0.87    19
4 -0.46  0.51    98
5 -0.97 -1.34    79

> vpts + vpts
      x     y value
1  1.31  0.64    14
2  1.15 -0.31   158
3 -1.15  0.87    38
4 -0.46  0.51   196
5 -0.97 -1.34   158
```

Defining Methods for Comparison Operators

A similar definition will work for the `Compare` group.

```
> setMethod("Compare", signature(e1 = "vcoords",
                                 e2 = "vcoords"),
      function(e1, e2)
      {
          if (!sameloc(e1, e2))
              stop("identical locations required")
          callGeneric(values(e1), values(e2))
      })
[1] "Compare"
```

Notice that this returns a logical vector rather than a `vcoords` object.

Additional Definitions

- The definitions of the binary operators given above only work for combining two `vcoords` objects.
- It may also be useful to define operations like `x + 10` or `x > 3`.
- This can be done by defining addition methods for combining `vcoords` objects and numeric objects.
- Care must be taken to make this work correctly.

Additional Methods

The following method will make expressions like `1 + x` and `3 * y` work correctly.

```
> setMethod("Arith",
      signature(e1 = "numeric",
                e2 = "vcoords"),
     function(e1, e2) {
         if (length(e1) > length(values(e2)))
             stop("incompatible lengths")
         vcoords(xcoords(e2),
                 ycoords(e2),
                 callGeneric(as.vector(e1),
                             values(e2)))
        })
[1] "Arith"
```

Example: Scaling `vcoords`

```
> vpts
      x     y value
1  1.31  0.64     7
2  1.15 -0.31    79
3 -1.15  0.87    19
4 -0.46  0.51    98
5 -0.97 -1.34    79

> 3 * vpts
      x     y value
1  1.31  0.64    21
2  1.15 -0.31   237
3 -1.15  0.87    57
4 -0.46  0.51   294
5 -0.97 -1.34   237
```

Additional Methods

The following method will make expressions like `1 + x` and `3 * y` work correctly.

```
> setMethod("Arith",
      signature(e1 = "vcoords",
                e2 = "numeric"),
      function(e1, e2) {
          if (length(values(e1)) < length(e2))
              stop("incompatible lengths")
          vcoords(xcoords(e1),
                  ycoords(e1),
                  callGeneric(values(e1),
                              as.vector(e2)))
      })
[1] "Arith"
```

Example: Scaling `vcoords`

```
> vpts
      x     y value
1  1.31  0.64     7
2  1.15 -0.31    79
3 -1.15  0.87    19
4 -0.46  0.51    98
5 -0.97 -1.34    79

> vpts / 2
      x     y value
1  1.31  0.64   3.5
2  1.15 -0.31  39.5
3 -1.15  0.87   9.5
4 -0.46  0.51  49.0
5 -0.97 -1.34  39.5
```

Example: `vcoords` Powers

```
> vpts
      x     y value
1  1.31  0.64     7
2  1.15 -0.31    79
3 -1.15  0.87    19
4 -0.46  0.51    98
5 -0.97 -1.34    79

> vpts^2
      x     y value
1  1.31  0.64    49
2  1.15 -0.31  6241
3 -1.15  0.87   361
4 -0.46  0.51  9604
5 -0.97 -1.34  6241
```

Testing Class Membership

The function `is` allows us to test whether an object belongs to a particular class.

```
> is(vpts, "vcoords")
[1] TRUE
```

Remember that the `vcoords` class is defined as inheriting from the `coords` class. So every `vcoords` object is also a `coords` object.

```
> is(vpts, "coords")
[1] TRUE
```

Coercion of Objects

Class inheritance provides a natural way of coercing objects from one class to another. The function `as` can be used to do this.

```
> as(vpts, "coords")
      x     y
1  1.31  0.64
2  1.15 -0.31
3 -1.15  0.87
4 -0.46  0.51
5 -0.97 -1.34
```

Coercion only works in the direction of inheritance (it is easy to discard slots).

Subsetting

It is likely that we will want to take subsets of `coords` and `vcoords` objects. We can do this by defining methods for the `[` generic.

```
> setMethod("[",
            signature(x = "vcoords",
                      i = "ANY",
                      j = "missing",
                      drop = "missing"),
            function(x, i, j)
            vcoords(xcoords(x)[i],
                    ycoords(x)[i],
                    values(x)[i]))
[1] "["
```

Example

```
> vpts[1:3]
      x     y value
1  1.31  0.64     7
2  1.15 -0.31    79
3 -1.15  0.87    19

> vpts[values(vpts) > 50]
      x     y value
1  1.15 -0.31    79
2 -0.46  0.51    98
3 -0.97 -1.34    79
```