Object oriented programming with R (S3 classes)

Dr Colin Gillespie

School of Mathematics & Statistics

February 14, 2014

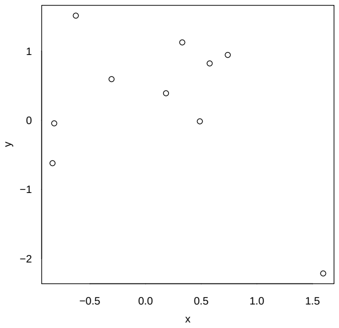
Example 1: The `plot` function

```
R> x = rnorm(10); y = rnorm(10)
R> m = lm(y ~ x)
```

Example 1: The plot function

```
R> x = rnorm(10); y = rnorm(10)
R> m = lm(y ~ x)
```

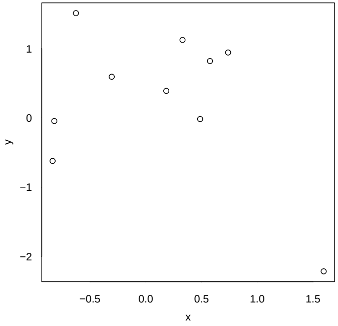
```
R> plot(x, y)
```



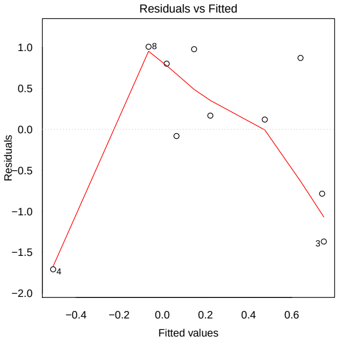
Example 1: The plot function

```
R> x = rnorm(10); y = rnorm(10)
R> m = lm(y ~ x)
```

```
R> plot(x, y)
```



```
R> plot(m)
```



Example 2: The summary function

```
R> summary(x)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -0.836  -0.546   0.257   0.132   0.554   1.600
```

Example 2: The summary function

```
R> summary(x)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  -0.836  -0.546   0.257   0.132   0.554   1.600
```

```
R> summary(m)
##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -1.708 -0.610  0.143  0.854  1.008
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    0.317      0.338    0.94     0.38
## x             -0.516      0.449   -1.15     0.28
```

OOP overview

Example: a simplified coda class

- Parallel runs of the same chain
- Different seeds and starting values
- Same number of iterations in each chain

OOP overview

Example: a simplified coda class

- Parallel runs of the same chain
- Different seeds and starting values
- Same number of iterations in each chain

- We could create a single object that contains all chains
  - A *class* is the formal definition of an object
  - When we create an individual object, we call this an *instance*
  - If a function operates on specific classes, this is called a method
- A user doesn't need to know how we've implemented the class - *encapsulation*
- A method can operate on multiple classes, e.g. `plot` - *polymorphism*

Overview of R's OOP

- The easiest and oldest system is the S3 class (generic-function OOP)
  - This type of OO is different to message-passing style of Java and C++
  - In a message-passing framework, messages/methods are sent to objects and the object determines which function to call - `normal.rand(1)`
  - In S3, the generic function decides which method to call - it has the form `rand(normal, 1)`

Overview of R's OOP

- The easiest and oldest system is the S3 class (generic-function OOP)
    - This type of OO is different to message-passing style of Java and C++
    - In a message-passing framework, messages/methods are sent to objects and the object determines which function to call - `normal.rand(1)`
    - In S3, the generic function decides which method to call - it has the form `rand(normal, 1)`
- The S4 system is a formal version of S3. The largest difference is that S4 system has formal class definitions
    - Bioconductor packages use S4 classes

Overview of R's OOP

- The easiest and oldest system is the S3 class (generic-function OOP)
    - This type of OO is different to message-passing style of Java and C++
    - In a message-passing framework, messages/methods are sent to objects and the object determines which function to call - `normal.rand(1)`
    - In S3, the generic function decides which method to call - it has the form `rand(normal, 1)`
- The S4 system is a formal version of S3. The largest difference is that S4 system has formal class definitions
    - Bioconductor packages use S4 classes
- Reference classes are different to S3 and S4. Reference classes use message passing and also have mutable states
    - Reference classes are just S4 objects with a fancy environment

S3 classes

- S3 classes are informal and simple to construct
- An S3 class, is just an object attribute

```
R> x = 5
R> class(x)

## [1] "numeric"
```

S3 classes

- S3 classes are informal and simple to construct
- An S3 class, is just an object attribute

```
R> x = 5
R> class(x)

## [1] "numeric"
```

- We can also change/alter an object's class

```
R> ## Multiple classes
R> class(x) = c("numeric", "myclass")
R> class(x)

## [1] "numeric" "myclass"
```

- R does not perform any sort of type checking.

Example: MCMC chains

- Sample MCMC output from a simple linear regression model given in the BUGS manual

```
R> head(chain1, 2)

##       alpha   beta  sigma
## [1,] 7.173 -1.566 11.233
## [2,] 2.953  1.503  4.886
```

Example: MCMC chains

- Sample MCMC output from a simple linear regression model given in the BUGS manual

```
R> head(chain1, 2)

##       alpha   beta  sigma
## [1,] 7.173 -1.566 11.233
## [2,] 2.953  1.503  4.886
```

- Store each chain as an element of a list

```
R> chains = list(2)
R> chains[[1]] = chain1
R> chains[[2]] = chain2
```

Example: MCMC chains

- Sample MCMC output from a simple linear regression model given in the BUGS manual

```
R> head(chain1, 2)

##       alpha   beta  sigma
## [1,] 7.173 -1.566 11.233
## [2,] 2.953  1.503  4.886
```

- Store each chain as an element of a list

```
R> chains = list(2)
R> chains[[1]] = chain1
R> chains[[2]] = chain2
```

- Update the class

```
R> class(chains) = "mymcmc"
```

Overloading existing generic methods

1. To determine if an existing function is a generic, use the `methods` function:

```
R> methods("t")

## [1] t.data.frame t.default    t.trellis*
## [4] t.ts*
##
##    Non-visible functions are asterisked
```

Overloading existing generic methods

1. To determine if an existing function is a generic, use the `methods` function:

```
R> methods("t")

## [1] t.data.frame t.default    t.trellis*
## [4] t.ts*
##
##    Non-visible functions are asterisked
```

2. To overload an existing generic function, we just create a function with name `generic.class`
3. We should match the arguments of the existing generic

Example: adding a plot method

Example: adding a plot method

- Check the arguments of `plot`

```
R> args(plot)

## function (x, y, ...)
## NULL
```

Example: adding a plot method

- Check the arguments of `plot`

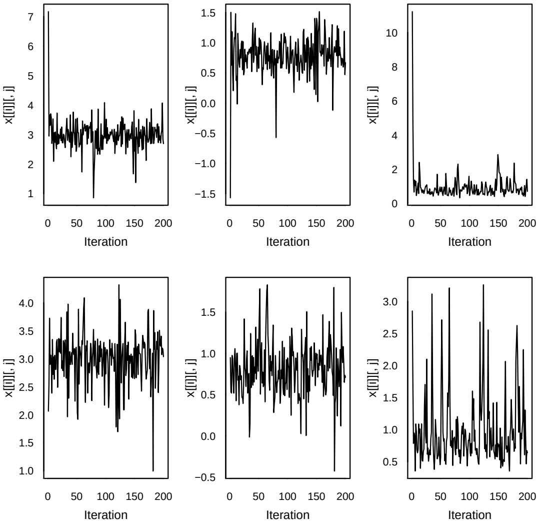
```
R> args(plot)

## function (x, y, ...)
## NULL
```

- Create a method for `mymcmc` objects

```
R> plot.mymcmc  = function(x, y, ...) {
+   n_chains = length(x); n_pars = ncol(x[[1]])
+   par(mfrow = c(n_chains, n_pars))
+   for(i in 1:n_chains) {
+     for(j in 1:n_pars) {
+       plot(x[[i]][, j], ...)
+     }
+   }
+ }
```

Example: adding a plot method

```
R> plot(chains, type = "l", xlab = "Iteration")
```



Creating generics from scratch: step 1

Creating generics from scratch: step 1

Create a base generic

```
R> sd = function(x, na.rm = FALSE, ...) UseMethod("sd")
```

The `UseMethod()` function takes two arguments.

1. The generic function name - in the above example this would be `sd`
2. The argument to use for method dispatch. If the second argument is omitted, the default is the first argument of the enclosing function. In this example, it would be `class(x)`.

Creating generics from scratch: step 2

Create a `default` method - `generic.default`

```
R> sd.default = function(x, na.rm = FALSE, ...)
+   stats::sd(x, na.rm=na.rm)
```

This is an optional step, but usually a good idea.

Creating generics from scratch: step 3

Create a class method

```
R> sd.mymcmc = function(x, na.rm = FALSE, ...)
+   lapply(x, apply, 2, function(i) sd(i, na.rm=na.rm, ...))
```

Creating generics from scratch: step 3

Create a class method

```
R> sd.mymcmc = function(x, na.rm = FALSE, ...)
+   lapply(x, apply, 2, function(i) sd(i, na.rm=na.rm, ...))
```

```
R> sd(chains)
## [[1]]
##  alpha   beta  sigma
## 0.5314 0.3406 0.8893
##
## [[2]]
##  alpha   beta  sigma
## 0.4643 0.3331 0.5572
```

A few bits and pieces

Finding methods

To find S3 methods associated with a particular function – use the `methods` function:

```
R> methods("plot")
##  [1] plot.acf*           plot.data.frame*
##  [3] plot.decomposed.ts* plot.default
##  [5] plot.dendrogram*    plot.density
##  [7] plot.ecdf           plot.factor*
##  [9] plot.formula*       plot.function
## [11] plot.hclust*        plot.histogram*
## [13] plot.HoltWinters*   plot.isoreg*
## [15] plot.lm             plot.mcmc*
## [17] plot.mcmc.list*     plot.medpolish*
## [19] plot.mlm            plot.mymcmc
## [21] plot.ppr*           plot.prcomp*
## [23] plot.princomp*      plot.profile.nls*
## [25] plot.shingle*       plot.spec
## [27] plot.stepfun        plot.stl*
```

Finding methods

If we wanted to show methods for a particular class, then we specify the class of interest

```
R> methods(class = "mymcmc")
## [1] plot.mymcmc sd.mymcmc
```

Function definitions

If we type the function name

```
R> plot
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x30242e8>
## <environment: namespace:graphics>
```

we just get the generic definition.

Function definitions

If we type the function name

```
R> plot
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x30242e8>
## <environment: namespace:graphics>
```

we just get the generic definition. To view a particular function definition use `getS3method`

```
R> getS3method("plot", "mymcmc")
## function(x, y, ...) {
##   n_chains = length(x); n_pars = ncol(x[[1]])
##   par(mfrow = c(n_chains, n_pars))
##   for(i in 1:n_chains) {
##     for(j in 1:n_pars) {
##       plot(x[[i]][, j], ...)
##     }
```

Avoid using `.` in variable names

Unfortunately a number of functions use `.` as a variable/function name. For example,

- There isn't a `csv` class - `read.csv`

Avoid using `.` in variable names

Unfortunately a number of functions use `.` as a variable/function name. For example,

- There isn't a `csv` class - `read.csv`

Even more confusing is the `t.test` function

Avoid using `.` in variable names

Unfortunately a number of functions use `.` as a variable/function name. For example,

- There isn't a `csv` class - `read.csv`

Even more confusing is the `t.test` function

```
R> z = t.test(rnorm(y))
R> class(z)
## [1] "htest"
```

Common sense: don't use `.` in variable names

NextMethod

It is typical for a method function to make a few changes to its arguments and dispatch to the next method. In this scenario, use `NextMethod`

```
R> t.data.frame = function(x) {
+     x = as.matrix(x)
+     NextMethod("t")
+ }
```

Inherent classes

Unfortunately the whole class system in R is a bit of mess. For example, suppose we create a single element vector

```
R> x = "animal"
```

and investigate it's class

```
R> class(x)
## [1] "character"
```

Inherent classes

Unfortunately the whole class system in R is a bit of mess. For example, suppose we create a single element vector

```
R> x = "animal"
```

and investigate it's class

```
R> class(x)
## [1] "character"
```

```
R> is(x, "character")
## [1] TRUE
```

Inherent classes

Unfortunately the whole class system in R is a bit of mess. For example, suppose we create a single element vector

```
R> x = "animal"
```

and investigate it's class

```
R> class(x)
## [1] "character"
```

```
R> is(x, "character")
## [1] TRUE
```

```
R> is.character(x)
## [1] TRUE
```

Inherent classes

Let’s change the class

```
R> class(x) = "animal"
```

Inherent classes

Let’s change the class

```
R> class(x) = "animal"
```

```
R> is(x, "character")
## [1] FALSE
```

Inherent classes

Let's change the class

```
R> class(x) = "animal"
```

```
R> is(x, "character")
## [1] FALSE
```

```
R> is.character(x)
## [1] TRUE
```

Remember, everything is function

```
R> "+" = function(e1, e2) e1 - e2
```

```
R> 3 + 2
## [1] 1
```

Any questions?