3. The object system(s)

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2017

Generics and methods

Many functions in R are generic. This means that the function itself (eg `plot`, `summary`, `mean`) doesn't do anything. The work is done by methods that know how to plot, summarize or average particular types of information.

If you call `summary` on a `data.frame`, R works out that the correct function to do the work is `summary.data.frame` and calls that instead. If there is no specialized method to summarize the information, R will call `summary.default`

You can find out all the types of data that R knows how to summarize with two functions...

Generics and methods

```
> methods("summary")
 [1] summary.Date           summary.POSIXct        summary.POSIXlt
 [4] summary.aov            summary.aovlist        summary.connection
 [7] summary.data.frame     summary.default        summary.ecdf*
[10] summary.factor         summary.glm            summary.infl
[13] summary.lm             summary.loess*         summary.manova
[16] summary.matrix         summary.mlm            summary.nls*
[19] summary.packageStatus* summary.ppr*           summary.prcomp*
[22] summary.princomp*      summary.stepfun        summary.stl*
[25] summary.table          summary.tukeysmooth*

    Non-visible functions are asterisked
> showMethods("summary")
NULL
```

There are two functions because S has two object systems, for historical reasons.

Generics and methods

Use the `class` argument to see which generics are available

```
> methods(class="lm")
 [1] add1.lm*           alias.lm*
 [3] anova.lm           case.names.lm*
 [5] confint.lm*        cooks.distance.lm*
 [7] deviance.lm*       dfbeta.lm*
 [9] dfbetas.lm*        drop1.lm*
[11] dummy.coef.lm*     effects.lm*
[13] extractAIC.lm*     family.lm*
[15] formula.lm*        hatvalues.lm
[17] influence.lm*      kappa.lm
```

... and many more; packages you load may have their own generics

Methods

The class and method system makes it easy to add new types of information (e.g. survey designs) and have them work just like the built-in ones.

Some standard methods are

- `print`, `summary`: everything should have these
- `plot` or `image`: if you can work out an obvious way to plot the thing, one of these functions should do it.
- `coef`, `vcov`: Anything that estimates parameters and corresponding covariance matrices should have these.
- `anova`, `logLik`, `AIC`: models fitted by maximum likelihood should have these.
- `residuals`: anything that has residuals should have this.

[Informal analogue of Java interfaces]

New classes: S3

Creating a new class is easy

```
class(x) <- "duck"
```

R will now automatically look for the `print.duck` method, the `summary.duck` method, and so on.

There is no formal registration or documentation of the structure of the object. *You* need to make sure that anything of class `duck` can `look.duck`, `walk.duck`, `quack.duck`.

Yes, this is different from Java and C++.

Generic functions: S3

A generic function has a call to `UseMethod()`, which does the method dispatching.

```
> print
function (x, ...)
{
UseMethod("print")
}
```

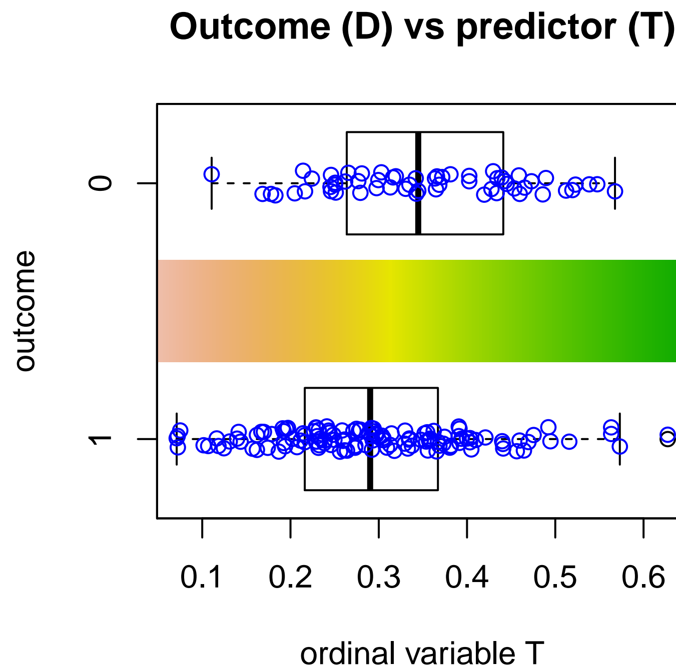
By default, method dispatch is on the first argument. It can be on any (single) argument.

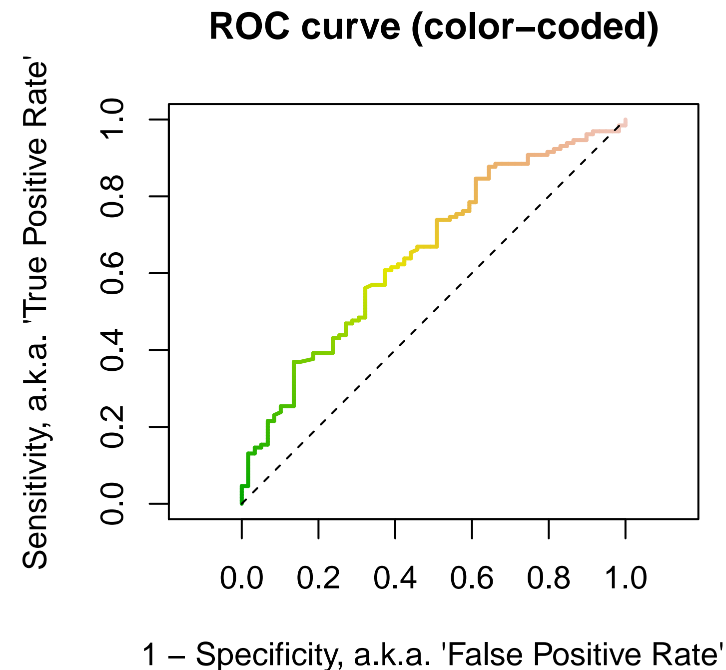
```
> svymean
function (x, design, na.rm = FALSE, ...)
{
    UseMethod("svymean", design)
}
```

Example: ROC curves

The Receiver Operating Characteristic (ROC) curve describes the ability of an ordinal variable T to predict a binary variable D.

The ROC curve graphs $P(T > c|D = 1)$ against $P(T > c|D = 0)$ for every cutpoint c;





Example: ROC curves

Here's a simple way to code it:

```
ROC <- function(test, disease){ #test = e.g. levels of a biomarker
  # where is the curve going to change?
  cutpoints <- c(-Inf, sort(unique(test)), Inf)

  # what values will it take when it does change?
  sensitivity <- sapply(cutpoints,
      function(result){ mean(test>result & disease)/mean(disease)}
      )

  specificity <- sapply(cutpoints,
      function(result){mean(test<=result & !disease)/mean(!disease)}
      )

  # plot the curve, return the coordinates
  plot(1-specificity, sensitivity, type="l")
  abline(0,1,lty=2)

  return(list(sens=sensitivity, spec=specificity))
}
```

Example: ROC curve

Here's a more efficient version of the calculation;

```
drawROC<-function(T,D){
  DD <- table(-T,D)
  tpr <- cumsum(DD[,2])/sum(DD[,2])
  fpr <- cumsum(DD[,1])/sum(DD[,1])
  plot(fpr, tpr, type="l")
}
```

Note that we use the vectorized `cumsum()` rather than the implied loop of `sapply()`.

We want to make this return an ROC object that can be plotted and operated on in other ways

ROC curve object

```
ROC<-function(T,D){
  DD <- table(-T,D)
  tpr <- cumsum(DD[,2])/sum(DD[,2])
  fpr <- cumsum(DD[,1])/sum(DD[,1])
  rval <- list(tpr=tpr, fpr=fpr,
               cutpoints=rev(sort(unique(T))),
               call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Instead of plotting the curve we return the data needed for the plot – plus some things that might be useful later; `sys.call()` is a copy of the call.

Methods

We need a `print` method to stop the whole contents of the object being printed

```
print.ROC<-function(x,...){
   cat("ROC curve: ")
   print(x$call)
}
```

Methods

A plot method

```
plot.ROC <- function(x, xlab="1-Specificity",
                         ylab="Sensitivity", type="l",...){
        plot(x$fpr, x$tpr, xlab=xlab, ylab=ylab, type=type, ...)
}
```

We specify some graphical parameters in order to set defaults for them. Others are automatically included in `...`.

Methods

We want to be able to add lines to an existing plot

```
lines.ROC <- function(x, ...){
lines(x$fpr, x$tpr, ...)
}
```

and also be able to identify cutpoints by clicking on a graph

```
identify.ROC<-function(x, labels=NULL, ...,digits=1)
{
  if (is.null(labels))
    labels<-round(x$cutpoints,digits)
  identify(x$fpr, x$tpr, labels=labels,...)
}
```

Syntax notes

Methods should have at least the same arguments as the generic, in the same order, with the same defaults (so the first argument to a `print` method is `x`, but to a `summary` method is `object`).

For inheritance to work, methods must have a `...` argument to allow unknown arguments to be ignored.

The language does not enforce these requirements, but the package checking system does.

Inheritance

The `class` attribute can be a vector, e.g. `c("glm", "lm")`

R will look for a method for each element in turn until it finds one.

Inside a method, use `NextMethod()` to call the next method in the inheritance.

Inheritance is not used much: statisticians extend by generalization, not by specialization. The relationship of `glm` to `lm` should really be delegation, not inheritance.

An exception is data infrastructure (e.g. Bioconductor), which tends to use S4 methods.

S4 classes

Introduced in version 4 of Bell Labs' S, since extended and refined in R.

Still uses generic functions, with methods belonging to functions rather than to classes.

- Formal declaration of class structure: `setClass()`
- Formal declaration of methods: `setMethod()`
- Multiple dispatch
- Multiple inheritance

Example: ROC curve

Define `ROC` class

```
setClass("ROC",
        representation(tpr="numeric",fpr="numeric",
           cutpoints="numeric",call="call")
)
```

Or we could factor out the 'curve' structure and declare

```
setClass("xycurve", representation(x="numeric", y="numeric"))
setClass("ROC",  contains="xycurve",
        representation(cutpoints="numeric",call="call")
)
```

taking advantage of inheritance

Example: ROC curve

Other options include validity checks at object creation

```
setClass("ROC",
     representation(tpr="numeric",fpr="numeric",
          cutpoints="numeric",call="call"),
     validity=function(object){
           if(length(object@tpr)!=length(object@fpr) ||
              length(object@tpr)!=length(object@cutpoints))
                  return("length mismatch")
           if(any(object@tpr>1) || any(object@fpr>1) ||
              any(object@tpr<0) || any(object@fpr<0))
                 return("outside [0,1]")
           return(TRUE)
        })
```

Example: ROC constructor

Objects are created with `new()`; code is otherwise the same.

```
ROC <- function(T,D){
        DD <- table(-T,D)
        tpr <- cumsum(DD[,2])/sum(DD[,2])
        fpr <- cumsum(DD[,1])/sum(DD[,1])
        new("ROC",tpr=tpr, fpr=fpr,
           cutpoints=rev(sort(unique(T))),call=sys.call())
        }
```

Example: ROC methods

`setMethod` specifies a method for a generic function and an argument signature giving the classes of all the arguments used for dispatch.

Use `@` to refer to slots (not `$`), otherwise similar to S3

```
setMethod("show",signature="ROC",
    function(object){
      cat("S4 ROC curve:")
      print(object@call)
    }
)
```

(Note that S4 uses `show` rather than `print`)

This generic has only one argument, so the signature is a single string.

Example: ROC methods

```
setMethod("plot",signature("ROC","ANY"),
   function(x,y,type="l", xlab="1 - Specificity",
            ylab="Sensitivity",...){
       plot(1-x@fpr, x@tpr, type=type, xlab=xlab, ylab=ylab ,...)
    }
)
```

This generic, for `plot()`, has two arguments (x, y).

The signature specifies this method when x is ROC and y is ANYthing.

Example: ROC methods

`lines()` is *not* an S4 generic, but we can re-use the S3 version;

```
setGeneric("lines")
setMethod("lines",signature("ROC"),
    function(x,...){
        lines(1-x@spec, x@sens,...)
    }
)
```

`setGeneric()` creates an S4 generic that defaults to calling the original `lines()` function.

Multiple dispatch

Generic functions with method choice based on all arguments are strictly more expressive than the Java/C++ model of methods belonging to classes.

Java/C++ style can be translated mechanically:
`object.method(arg1, arg2)` maps to `generic(object, arg1, arg2)`

The price is slower method lookup, but most of the cost is at installation time, and slower method lookup is inevitable for a system that allows one package to declare methods for another package's objects.

Multiple dispatch

Generic function style ;

- allows symmetric treatment of argument, e.g. matrix multiplication: `multiply(A, B)` not `A.rightmultiply(B)` or `B.leftmultiply(A)`

- allows the programmer to describe whether methods for two objects are actually doing the same thing.

- allows first-class functions, which mathematicians and statisticians like.

Multiple dispatch

`filter()` in the `flowCore` package for flow cytometry has two arguments: a data set, and an object specifying a subsetting operation. Methods are dispatched based on both arguments.

```
> showMethods("filter")
Function: filter (package flowCore)
x="flowFrame", filter="filter"
x="flowFrame", filter="filterSet"
x="flowSet", filter="filter"
x="flowSet", filter="filterList"
x="flowSet", filter="filterSet"
x="flowSet", filter="list"
```

The `Matrix` package has 70 multiplication methods for different combinations of matrix types (`showMethods("%*%")`)

More complex example

Class `AnnDbBimap` is used in the `AnnotationDbi` package in Bioconductor, to provide conversions from one system of identifiers to another (eg probe ids, gene ids, gene symbols, GO categories). More details in Session 9.

Examine the structure and inheritance relationships of the class with `getClass()`

More complex example

```
> getClass("AnnDbBimap")
Class "AnnDbBimap" [package "AnnotationDbi"]
Slots:
Name:      L2Rchain    direction        Lkeys        Rkeys   ifnotfound    datacache
Class:         list      integer    character    character         list environment
Name:    objName    objTarget
Class:  character  character

Extends:
Class "Bimap", directly
Class "AnnDbObj", directly
Class "AnnObj", by class "AnnDbObj", distance 2

Known Subclasses:
Class "InpAnnDbBimap", directly
Class "GoAnnDbBimap", directly
Class "GOTermsAnnDbBimap", directly
Class "AnnDbMap", directly
Class "ProbeAnnDbBimap", directly
Class "Go3AnnDbBimap", by class "GoAnnDbBimap", distance 2
Class "IpiAnnDbMap", by class "AnnDbMap", distance 2
Class "AgiAnnDbMap", by class "AnnDbMap", distance 2
Class "ProbeAnnDbMap", by class "AnnDbMap", distance 2    [..etc..]
```

More complex example

```
setClass("AnnDbBimap",
    contains=c("Bimap", "AnnDbObj"),
    representation(
        L2Rchain="list",              # list of L2Rlink objects
        direction="integer",          # 1L for left-to-right,
        Lkeys="character",
        Rkeys="character",
        ifnotfound="list"
    ),
    prototype(
        direction=1L,                 # left-to-right by default
        Lkeys=as.character(NA),
        Rkeys=as.character(NA),
        ifnotfound=list()             # empty list => raise an error
    )
)
```

More complex example

Multiple inheritance used for 'mix-in' behavior:

- `"Bimap"` is a virtual class that is used only to define a set of methods for its subclasses
- Some implementation is inherited from `"AnnDBObj"`

is(), as()

- `is(object, "class")` tests whether `object` inherits from `"class"`
- `as(object, "class")` attempts to convert `object` to `"class"`. This will only work if `object` inherits from `"class"` or a conversion function has been provided with `setAs()`

```
setAs("ROC", "numeric",
   function(from){ cbind(from@fpr, from@tpr, from@cutpoints) }
)
```

New generics

When creating a completely new function with methods, you need to specify the arguments to the generic function:

```
setGeneric("increment",
  function(object, step, ...)
    standardGeneric("increment")
)
```

Recall in S3 we'd have just defined `increment.ROC`, `increment.lm`, etc, which a generic `increment()` function would pick from with `UseMethod("increment")`

In S4, methods for `increment` will have a signature specifying classes for `object` and `step`

Some Bioconductor infrastructure

- `eSet`: basic data structure including genomic data, phenotype, metadata; specializes to `ExpressionSet`, `SnpSet`, others
- `IRanges`: for manipulating numeric sequences.
- `Xstring`: stores long strings (specializes to `DNAstring`, `RNAstring`, `AAstring`)
- `AnnDbObj`, `Bimap`: Storage and lookup of annotation data

eSet

assayData Contains matrices with equal dimensions, and with column number equal to `nrow(phenoData)`. Class:AssayData-class

phenoData Contains experimenter-supplied variables describing sample (i.e., columns in assayData) phenotypes. Class: AnnotatedDataFrame-class

featureData Contains variables describing features (i.e., rows in assayData) unique to this experiment. Use the annotation slot to efficiently reference feature data common to the annotation package used in the experiment. Class: AnnotatedDataFrame-class

experimentData Contains details of experimental methods. Class: MIAME-class

annotation Label associated with the annotation package used in the experiment. Class: character

protocolData Contains microarray equipment-generated variables describing sample (i.e., columns in assayData) phenotypes. Class: AnnotatedDataFrame-class

eSet

`eSet` has accessor functions to extract or modify the data; the slots should not be used directly.

`eSet` is a virtual class that abstracts a set of data properties. Actual objects must be defined using a subclass of `eSet`, and `new("eSet")` is an error.

`ExpressionSet` is a subclass where the `assayData` slot contains one or more matrices (all the same size) for gene expression data

`SnpSet` is a subclass where the `assayData` slot contains two matrices of the same size, for SNP calls and call probabilities

Sequences

The `IRanges` package provides an alternative infrastructure to vectors, mostly as virtual classes.

- `Sequence`: virtual class for (potentially large) vectors
- `View`: virtual class for subsequences of a `Sequence`
- `Ranges`: sets of intervals of consecutive integers.
- `IntervalTree`: find overlaps between two `Ranges`

Biostrings package

DNAString and RNAString represent genomic sequences, AAString represents an amino-acid sequence

```
>  d <- DNAString("TTGAAAA-CTC-N")
>   length(d)
[1] 13
>   alphabet(d)                  # DNA_ALPHABET
 [1] "A" "C" "G" "T" "M" "R" "W" "S" "Y" "K" "V" "H" "D" "B" "N" "-" "+"
>   alphabet(d, baseOnly=TRUE)  # DNA_BASES
[1] "A" "C" "G" "T"
>
> d
  13-letter "DNAString" instance
seq: TTGAAAA-CTC-N
> reverseComplement(d)
  13-letter "DNAString" instance
seq: N-GAG-TTTTCAA
> RNAString(d)
  13-letter "RNAString" instance
seq: UUGAAAA-CUC-N
```

Efficiency

The underlying sequence is not copied on assignment.

The `subseq()` function (from `IRanges`) makes a view of a subset of the string without copying

...allows manipulation of whole-chromosome sequences.

```
>   data(yeastSEQCHR1)
>   yeast1 <- DNAString(yeastSEQCHR1)
> str(yeast1)
Formal class 'DNAString' [package "Biostrings"] with 6 slots
  ..@ shared         :Formal class 'SharedRaw' [package "IRanges"] with 2 slots
  .. .. ..@ xp                    :<externalptr>
  .. .. ..@ .link_to_cached_object:<environment: 0x1cf45fdc>
  ..@ offset         : int 0
  ..@ length         : int 230208
  ..@ elementMetadata: NULL
  ..@ elementType    : chr "ANY"
  ..@ metadata       : list()
```

Efficiency

```
>   dinucleotideFrequency(yeast1)
   AA     AC     AG     AT     CA     CC     CG     CT     GA
23947  12493  13621  19769  15224   9218   7089  13112  14478
   GC     GG     GT     TA     TC     TG     TT
 8910   9438  12938  16181  14021  15617  24151
>   trinucleotideFrequency(yeast1)
 AAA  AAC  AAG  AAT  ACA  ACC  ACG  ACT  AGA  AGC  AGG
8576 4105 4960 6306 3924 2849 2186 3534 4537 2680 2707
 AGT  ATA  ATC  ATG  ATT  CAA  CAC  CAG  CAT  CCA  CCC
3697 5242 3849 4294 6384 5147 2722 3091 4264 3696 1622
 CCG  CCT  CGA  CGC  CGG  CGT  CTA  CTC  CTG  CTT  GAA
1444 2456 2158 1380 1446 2105 2755 2556 3074 4727 5437
 GAC  GAG  GAT  GCA  GCC  GCG  GCT  GGA  GGC  GGG  GGT
2384 2645 4012 2993 1960 1259 2698 2983 1905 1594 2955
 GTA  GTC  GTG  GTT  TAA  TAC  TAG  TAT  TCA  TCC  TCG
3490 2455 2798 4195 4787 3282 2925 5187 4611 2786 2200
 TCT  TGA  TGC  TGG  TGT  TTA  TTC  TTG  TTT
4424 4800 2945 3691 4181 4694 5161 5451 8845
```

Efficiency

```
>   ## Get the least and most represented 6-mers:
>   f6 <- oligonucleotideFrequency(yeast1, 6)
>   f6[f6 == min(f6)]
CCCGGG
     3
>   f6[f6 == max(f6)]
TTTTTT
   705
```

Comparisons

The S3 system has less overhead, is more widely understood, and is very slightly faster. It is still useful for single-programmer work.

The S4 system is better for multi-person efforts or code that is likely to be reused by others.

- Formal definition of class structure, so the contents of an object can be relied on
- Registration of methods means that reflection (looking up what methods are available) is reliable
- Multiple inheritance is useful for mix-in behavior
- Multiple dispatch is only rarely important, but when you need it you really need it