Stat 3701 Lecture Notes: R Generic Functions

Charles J. Geyer

September 06, 2022

1 License

2 R

- The version of R used to make this document is 4.2.1.
- The version of the `rmarkdown` package used to make this document is 2.15.
- The version of the `MASS` package used to make this document is 7.3.58.1.
- The version of the `mgcv` package used to make this document is 1.8.40.
- The version of the `aster` package used to make this document is 1.1.2.
- The version of the `gmp` package used to make this document is 0.6.6.

3 OOP

We learned in Sections 3.6, 3.7, 3.8 of course notes about R basics that R is not an object-oriented programming (OOP) language in the sense that C++ or Java is, although it is more object-oriented in some other senses (in R *everything* is an object). But R objects are not like C++ or Java objects.

- Most are not *instances* of *classes*, that is R is not a *classical* OOP language (but R S4 object system is classical (has classes)).
- Most are not created by the `new` operator (but R S4 objects are).
- Most do not have both fields and methods (data and functions). Even most R S4 classes do not have methods. Methods are not the R way. Instead R has generic functions.

So R has some but not all of what you would think OOP is from exposure to C++. Actually C++ is not a pure OOP language in the sense that Java is. By its need to be backward compatible with C, which is definitely not an OOP language, C++ is only as OOPy as you want it to be.

As Bjarne Stroustrup the inventor of C++ says, C++ is not "just" an OOP language, rather it is a multiparadigm language, and generic programming (like R favors) is one of its paradigms.

4 S3 generics

4.1 Introduction

R has *a lot* of S3 generic functions. You do not really understand R until you know at least a little bit about them. The reason they are called "S3" is because they were introduced in S version 3 (recall that S was the proprietary language of which R is a free implementation). The reason why one has to keep mentioning "S3" is to distinguishing them from the S4 system, which was introduced in S version 4. (S versions have nothing to do with R versions. R is not up to version 4 yet. S got to version 4 long ago.)

You can tell an S3 generic just by looking at it.

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x556ad008a968>
## <environment: namespace:base>
```

```
summary
```

```
## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x556ad01e37c8>
## <environment: namespace:base>
```

The entire body of the function is a call to the R function `UseMethod`. Also the documentation

```
?plot
?summary
```

says the function is generic right at the beginning.

So called group generics are different; they will be discussed below.

4.2 S3 Classes

4.2.1 Introduction

To understand S3 generics, we first have to understand classes. S3 classes are *very* different from C++ classes. Basically, they mean nothing. To make `foo` an object of class `bar` you just say so.

```
foo <- 1:10
class(foo)
```

```
## [1] "integer"
```

```
class(foo) <- "bar"
class(foo)
```

```
## [1] "bar"
```

```
attributes(foo)
```

```
## $class
## [1] "bar"
```

So `foo` having class `bar` doesn't mean anything other than that someone set the `"class"` attribute of `foo` to be the character string `"bar"`.

Usually, one uses the R function `class` to do that, but one doesn't have to.

```
foo <- 1:10
attr(foo, "class") <- "bar"
class(foo)
```

```
## [1] "bar"
```

So an S3 class is just a character string attribute. It says nothing whatsoever about any other properties of the object.

4.2.2 Subclasses

The class attribute can be a character vector.

```
foo <- 1:10
class(foo) <- c("bar", "baz", "qux", "woof")
```

We can think of object `foo` having class `"woof"`, subclass `"qux"`, subsubclass `"baz"`, subsubsubclass `"bar"`.

Again this has nothing whatsoever to do with the properties of the object. The `"class"` attribute is whatever somebody set it to.

4.2.3 Basic R Objects

R objects have a class — the class of `foo` is given by `class(foo)` even if `foo` does not have a class attribute — the way this works is that the R function `class` makes up a class for those objects.

For some the class is the same as the mode.

```
class(1)
```

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

```
mode(1)
```

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

For matrices and arrays the class is determined by whether the object has an attribute `"dim"` and what its length is (recall that *really* a matrix is just a numeric vector with an attribute `"dim"` giving the dimensions and similarly for arrays).

4.2.4 Summary

- Everything in R is an object
- Every object has a class, whether or not it has been assigned one, and the R function `class` reports the class or vector of classes.

4.3 How Generics Work

S3 generics "dispatch" on the class of the first argument.

- They look at the class of the first argument.
- They paste the class onto the function separated by a dot.
- If a function of that name exists, they redo the function call with the same arguments but the new function name.
- If the R function `class` returns a vector when applied to the first argument, they try each component in turn until they find a match.
- If no matching function is found, the paste "default" onto the name of the generic function, and redo the function call with that function, if it exists.
- If no match is found, this is an error.

```
fred <- function(x) UseMethod("fred")
fred(1:4)
```

```
## Error in UseMethod("fred"): no applicable method for 'fred' applied to an object of class "c('intege
fred.default <- function(x) {
    cat("in fred.default\n")
    invisible(NULL)
}
fred(1:4)
```

```
## in fred.default
```

```
fred.foo <- function(x) {
    cat("in fred.foo\n")
    invisible(NULL)
}
o <- 1:4
class(o) <- "foo"
fred(o)
```

```
## in fred.foo
```

```
fred.bar <- function(x) {
    cat("in fred.bar\n")
    invisible(NULL)
}
class(o) <- c("foo", "bar")
fred(o)
```

```
## in fred.foo
```

The R function `structure` is designed to make and object and assign attributes to it in one go. So we can also do

```
fred(structure(1:4, class = "bar"))
```

```
## in fred.bar
```

In the terminology of S3 generics, `fred.foo`, `fred.bar`, and `fred.default` are called *methods* of the generic function `fred`.

4.4 Why Should We Care?

You are unlikely to ever want to write an R generic function until you have enough R knowledge to write a CRAN package. Then, if you want to write a regression-like package, you certainly need to consider adding generic functions that users expect to have for such packages, like those that have methods for class `"lm"`.

But even if you don't want to write CRAN packages or S3 generics, you will need to use them. So you need to know how to discover them.

All available methods for a generic function are given by the R function `methods`.

```
methods(summary)
```

```
##  [1] summary.aov                    summary.aovlist*
##  [3] summary.aspell*                summary.check_packages_in_dir*
##  [5] summary.connection             summary.data.frame
##  [7] summary.Date                   summary.default
##  [9] summary.ecdf*                  summary.factor
## [11] summary.glm                    summary.infl*
## [13] summary.lm                     summary.loess*
## [15] summary.manova                 summary.matrix
## [17] summary.mlm*                   summary.nls*
## [19] summary.packageStatus*              summary.POSIXct
## [21] summary.POSIXlt                     summary.ppr*
## [23] summary.prcomp*                     summary.princomp*
## [25] summary.proc_time                   summary.rlang_error*
## [27] summary.rlang_message*              summary.rlang_trace*
## [29] summary.rlang_warning*              summary.rlang:::list_of_conditions*
## [31] summary.srcfile                     summary.srcref
## [33] summary.stepfun                     summary.stl*
## [35] summary.table                       summary.tukeysmooth*
## [37] summary.warnings
## see '?methods' for accessing help and source code
```

There is also a function

```
.S3methods(summary)
```

```
##  [1] summary.aov                         summary.aovlist*
##  [3] summary.aspell*                     summary.check_packages_in_dir*
##  [5] summary.connection                  summary.data.frame
##  [7] summary.Date                        summary.default
##  [9] summary.ecdf*                       summary.factor
## [11] summary.glm                         summary.infl*
## [13] summary.lm                          summary.loess*
## [15] summary.manova                      summary.matrix
## [17] summary.mlm*                        summary.nls*
## [19] summary.packageStatus*              summary.POSIXct
## [21] summary.POSIXlt                     summary.ppr*
## [23] summary.prcomp*                     summary.princomp*
## [25] summary.proc_time                   summary.rlang_error*
## [27] summary.rlang_message*              summary.rlang_trace*
## [29] summary.rlang_warning*              summary.rlang:::list_of_conditions*
## [31] summary.srcfile                     summary.srcref
## [33] summary.stepfun                     summary.stl*
## [35] summary.table                       summary.tukeysmooth*
## [37] summary.warnings
## see '?methods' for accessing help and source code
```

that lists only S3 methods, if we want to bother with that. (In this case *all* the methods are S3.)

The result changes as packages are loaded.

```
library(MASS)
library(mgcv)
```

```
## Loading required package: nlme
```

```
## This is mgcv 1.8-40. For overview type 'help("mgcv-package")'.
```

```
library(aster)
methods(summary)
```

```
##  [1] summary,ANY-method                  summary,diagonalMatrix-method
##  [3] summary,sparseMatrix-method         summary.aov
##  [5] summary.aovlist*                    summary.aspell*
##  [7] summary.aster                       summary.check_packages_in_dir*
##  [9] summary.connection                  summary.corAR1*
## [11] summary.corARMA*                    summary.corCAR1*
## [13] summary.corCompSymm*                summary.corExp*
## [15] summary.corGaus*                    summary.corIdent*
## [17] summary.corLin*                     summary.corNatural*
## [19] summary.corRatio*                   summary.corSpher*
## [21] summary.corStruct*                  summary.corSymm*
## [23] summary.data.frame                  summary.Date
## [25] summary.default                     summary.ecdf*
## [27] summary.factor                      summary.gam
## [29] summary.glm                         summary.gls*
## [31] summary.infl*                       summary.lm
## [33] summary.lme*                        summary.lmList*
## [35] summary.loess*                      summary.loglm*
## [37] summary.manova                      summary.matrix
## [39] summary.mlm*                        summary.modelStruct*
## [41] summary.negbin*                     summary.nls*
## [43] summary.nlsList*                    summary.packageStatus*
## [45] summary.pdBlocked*                  summary.pdCompSymm*
## [47] summary.pdDiag*                     summary.pdIdent*
## [49] summary.pdIdnot*                    summary.pdLogChol*
## [51] summary.pdMat*                      summary.pdNatural*
## [53] summary.pdSymm*                     summary.pdTens*
## [55] summary.polr*                       summary.POSIXct
## [57] summary.POSIXlt                     summary.ppr*
## [59] summary.prcomp*                     summary.princomp*
## [61] summary.proc_time                   summary.reaster
## [63] summary.reStruct*                   summary.rlang_error*
## [65] summary.rlang_message*              summary.rlang_trace*
## [67] summary.rlang_warning*              summary.rlang:::list_of_conditions*
## [69] summary.rlm*                        summary.shingle*
## [71] summary.srcfile                     summary.srcref
## [73] summary.stepfun                     summary.stl*
## [75] summary.table                       summary.trellis*
## [77] summary.tukeysmooth*                summary.varComb*
## [79] summary.varConstPower*              summary.varConstProp*
## [81] summary.varExp*                     summary.varFixed*
## [83] summary.varFunc*                    summary.varIdent*
## [85] summary.varPower*                   summary.warnings
## see '?methods' for accessing help and source code
```

Following the comment that `methods` tacks on the end of its output, we look at

```
?methods
```

says that we need to say, for example,

```
?summary.lm
```

to see the help for the `lm` method for the generic function `summary`.

Although we say

```
summary(lout)
```

to invoke the `lm` method for the generic function `summary` when `lout` is an object of class `"lm"`, if we just say

```
?summary
```

we don't get the help for the method but rather the help for the generic function and perhaps a few methods for (classes of) basic R objects.

The `See Also` section of `?summary` says to look at help for `summary.lm` and `summary.glm` but we can see that there are a huge number of other summary methods, and only the R function `methods` tells us what they are.

In the printout of R function `methods`, a `*` following the name is not part of the name of a method, but indicates (according to `?methods`) that the method is not exported from the namespace of the package, which means for example that

```
summary.loglm
```

```
## Error in eval(expr, envir, enclos): object 'summary.loglm' not found
```

does not show you the source code for the method. You have to say instead

```
getS3method("summary", "loglm")
```

to see the source code.

One might hope that there is a function that does the reverse job of `methods`, that is, given a class, what generics have a method for it? But there does not seem to be such a function.

If one looks at the help for the R function that creates objects of this class (for example, `?lm` for objects of class `"lm"`), then the `See Also` section of the help should refer to all generics that have methods for this class. But there isn't an R function that does this job.

We can do it easily enough for one R package

```
myclass <- "lm"
myregex <- paste("\\.", myclass, "$", sep = "")
noo <- ls(envir = as.environment("package:stats"))
loo <- grep(myregex, noo, value = TRUE)
goo <- sub(myregex, "", loo)
sapply(goo, function(x) !is.null(getS3method(x, myclass, optional = TRUE)))
```

```
##      confint   dummy.coef model.matrix      predict    residuals      summary
##         TRUE         TRUE         TRUE         TRUE         TRUE         TRUE
```

There is some deep magic here we don't want to explain. The R object `regex` is a so-called regular expression which matches the string `".lm"` at the end of another string. The R function `grep` (as called here) returns the strings that match, and the R function `sub` (as called here) strips off the `".lm"` at then end, leaving us with the name of a putative generic function. We use the R function `getS3method` to tell us whether it really is an S3 method.

5 Group Generics

Recall that `+`, "[", "[<-", and other such basic syntax of R are also functions. They are also generic, behaving magically as both S3 and S4 generics.

If we do

```
get("+")
```

```
## function (e1, e2)  .Primitive("+")
```

it doesn't look like either an S3 or an S4 generic, but

```
methods("+")
```

```
## [1] +,dgTMatrix,dgTMatrix-method +,Matrix,missing-method
## [3] +.Date                       +.POSIXt
## see '?methods' for accessing help and source code
```

Finds some methods. Some of them are S3.

```r
.S3methods("+")
```

```
## [1] +.Date   +.POSIXt
## see '?methods' for accessing help and source code
```

There are several differences between group generics and other generics.

- You cannot tell they are generic by looking at their R source code. You have to look at the documentation.
- The help `?"+"` refers us to `?Ops` for how the group generic functions work.
- `?Ops` tells us that, *unlike other generics*, group generics for binary operators *dispatch on the classes of both operands*. We won't bother with those details. The mechanism usually does what one wants and gives a warning if it isn't clear what method should be called.

Rather than write our own methods for group generics, let us just see them in action in the CRAN package `gmp` which does (among other things) infinite precision rational arithmetic.

```r
library(gmp)
```

```
##
## Attaching package: 'gmp'

## The following objects are masked from 'package:base':
##
##     %*%, apply, crossprod, matrix, tcrossprod
```

```r
foo <- as.bigq(1:10, 11:20)
foo
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 1/11 1/6  3/13 2/7  1/3  3/8  7/17 4/9  9/19 1/2
```

```r
bar <- as.bigq(6:15, 16:25)
bar
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 3/8   7/17  4/9   9/19  1/2   11/21 6/11  13/23 7/12  3/5
```

```r
foo + bar
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 41/88   59/102  79/117  101/133 5/6     151/168 179/187 209/207 241/228
## [10] 11/10
```

```r
foo * bar
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 3/88   7/102  4/39   18/133 1/6    11/56  42/187 52/207 21/76  3/10
```

```r
foo / bar
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 8/33    17/42   27/52   38/63   2/3     63/88   77/102  92/117  108/133
## [10] 5/6
```

```r
set.seed(42)
as.bigq(rnorm(5))
```

```
## Big Rational ('bigq') object of length 5:
## [1] 3087123975855085/2251799813685248  -5086348948552451/9007199254740992
## [3] 1635384977986481/4503599627370496  2850159791879263/4503599627370496
## [5] 3641325338910997/9007199254740992
```

And now we show method dispatch working on both arguments as we would hope that it works.

```
foo + 1
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 12/11 7/6   16/13 9/7   4/3   11/8  24/17 13/9  28/19 3/2
```

```
1 + foo
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 12/11 7/6   16/13 9/7   4/3   11/8  24/17 13/9  28/19 3/2
```

```
foo^2
```

```
## Big Rational ('bigq') object of length 10:
##  [1] 1/121  1/36   9/169  4/49   1/9    9/64   49/289 16/81  81/361 1/4
```

```
2^foo
```

```
## Error in `^.bigq`(2, foo): <bigq> ^ <non-int>  is not rational; consider  require(Rmpfr); mpfr(*) ^
```

Note that in `1 + foo` it dispatched on the class `"bigq"` of the second argument, which is what we wanted (the default method for addition wouldn't know what to do with an argument of class `"bigq"`).

This package cannot deal with `2^foo` because (as the warning and error messages say) the result isn't rational. They refer you to another package that does arbitrary precision floating point.

6 Summary

This is as good a place as any to stop.

R OOP isn't much like C++ OOP. But it does provide very convenient functionality for users. Users can say `summary(foo)` and it works no matter what the class of `foo` is (there is always `summary.default` if no other method applies).

In some ways the OOP in R is even more powerful because

- Everything that exists is an object.
- Everything that happens is a function call.

(the quote from John Chambers referenced in the course notes on basics of R).

So generic functions in R can work on everything that exists and everything that happens. You can't say that about C++ style OOP.