The R language is peculiar in many ways, and its approach to object-oriented (OO) programming is just one of them. Indeed, base R supports not one, but three different OO systems: S3, S4 and RC classes. And yet, probably none of them would qualify as a fully-fledged OO system before the astonished gaze of an expert in languages such as Python, C++ or Java. In this tutorial, we will review the S3 system, the simplest yet most elegant of them. The use case of the quantities framework (CRAN packages units, errors and quantities) will serve as the basis to study the main concepts behind S3 programming in R: classes, generics, methods and inheritance.

**Introduction**

You probably heard of John Chambers’ slogans about R, i.e., *everything that exists is an object* and *everything that happens is a function call* (and a function is an object too). In most languages, *object* is an OO concept referring to an instance of a certain class that encapsulates its whole lifetime: construction, associated data, behavior and destruction. In R, instead, roughly speaking, an object is just a base type with possibly some attributes. In other words, everything is about the data in R, while construction, behavior and destruction are defined elsewhere.

Let us start with the common example: what are *factors* in R? What makes them so special?

x <- factor(c("apple", "orange", "orange", "pear"))

typeof(x)

## [1] "integer"

attributes(x)

## $levels

## [1] "apple" "orange" "pear"

##

## $class

## [1] "factor"

unclass(x)

## [1] 1 2 2 3

## attr(,"levels")

## [1] "apple" "orange" "pear"

As you can see, factors are just integer vectors with some metadata stored as an attribute named levels. Note also that there is a special attribute called class which is central in defining how these objects behave. Where and how to do it is what S3 programming is about. This decoupling makes S3 less formal and more error-prone, but on the other hand, it is much more flexible, because it allows you e.g. to extend classes in other packages without requiring them to modify a single bit.

Arguably, there are two main use cases for an S3-style design:

1. Certain procedure generates a list of heterogeneous things that needs to be subsequently manipulated as a whole. For example, the output from a linear regression:

fit <- lm(Petal.Width ~ Petal.Length, data=iris)

typeof(fit)

## [1] "list"

class(fit)

## [1] "lm"

names(fit)

## [1] "coefficients" "residuals" "effects" "rank"

## [5] "fitted.values" "assign" "qr" "df.residual"

## [9] "xlevels" "call" "terms" "model"

1. Certain base type requires to be enriched with some metadata that needs to be manipulated alongside it to provide some additional functionality (e.g., factors).

In this practical introduction to S3 programming, we will cover the second use case by reviewing the basic design of the quantities framework, which enables quantity calculus for R vectors, matrices and arrays, with automatic propagation, conversion, derivation and simplification of magnitudes and uncertainties.

**Defining a new object**

As a motivating example, we would like R to be able to automatically handle measurements with uncertainty (hereinafter, for convenience, just *errors*). Base numeric vectors are just fine to handle measurements; we just need to add the errors as metadata (in an attribute) and choose a proper class. In order to do this, no special care is needed beyond 1) avoiding special attribute names for our metadata, with special meaning (such as class, names, dim and dimnames), and 2) avoiding class names that are already in use out there.

In our case, we will store the errors in an attribute called errors, and the class name will be errors (not fanciful, just convenient). The first piece we need is a *constructor*:

set\_errors <- function(x, value=0) {

# check that 'value' has a proper length

stopifnot(any(length(value) == c(length(x), 1L)))

if (length(value) == 1)

value <- rep(value, length(x))

structure(x, errors=abs(value), class="errors")

# equivalent to:

# attr(x, "errors") <- abs(value)

# class(x) <- "errors"

# x

}

The attr and attributes functions allows us to retrieve attributes, but the user should not need to know the particular implementation details, so let us define an explicit *getter*:

errors <- function(x) attr(x, "errors")

Now, we can define a set of measurements and asign an error of 0.1 to each one of them as follows:

x <- set\_errors(1:10, seq(0.1, 1, 0.1))

x

## [1] 1 2 3 4 5 6 7 8 9 10

## attr(,"errors")

## [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

## attr(,"class")

## [1] "errors"

errors(x)

## [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

But still, nothing interesting happens with them.

x + x

## [1] 2 4 6 8 10 12 14 16 18 20

## attr(,"errors")

## [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

## attr(,"class")

## [1] "errors"

**Modifying the behavior: generics, methods and groups**

One of the first things we would probably want to do with our new object is to print it properly, because our users are not interested in the implementation details that we are able to see above (i.e., the actual attributes). Therefore, we need to provide a print method.

There are great sources to teach you all the technical details about S3 (there are some links at the end of this article). We will limit ourselves here to briefly introduce the basic building blocks of S3 programming: generics and methods. Many (most?) of the functions you daily use are *generics* (or *internal generics*, which are internally implemented functions that *mostly* behave like generics; see help(InternalMethods)).

isS3stdGeneric(print)

## print

## TRUE

isS3stdGeneric(mean)

## mean

## TRUE

.S3PrimitiveGenerics

## [1] "anyNA" "as.character" "as.complex" "as.double"

## [5] "as.environment" "as.integer" "as.logical" "as.numeric"

## [9] "as.raw" "c" "dim" "dim<-"

## [13] "dimnames" "dimnames<-" "is.array" "is.finite"

## [17] "is.infinite" "is.matrix" "is.na" "is.nan"

## [21] "is.numeric" "length" "length<-" "levels<-"

## [25] "names" "names<-" "rep" "seq.int"

## [29] "xtfrm"

This basically means that you can write *methods* for those generics to implement how they behave when they are applied to a particular object class. And this is as easy as writing a function named .. The signature of this function must accommodate, at least, the same arguments as the generic. In other words, a method can be *more specific* than the generic, but cannot lose generality. Some examples:

* For generic foo(x), method must be foo.bar(x).
* For generic foo(x, y, z), method can be foo.bar(x, y, z), or, if you are just interested in the first argument, for example, you may use dots foo.bar(x, ...) to summarize the rest of them.
* For generic foo(x, ...), method can be foo.bar(x, ...), or you may be more specific and take some arguments out of the dots foo.bar(x, y, z, ...) (see help(print) for examples).

In the following example, print.errors is defined. This particular implementation just shows a maximum of 5 error terms and then delegates the printing of the main vector of values to the default method by calling NextMethod.

print.errors <- function(x, ...) {

# print 5 errors at most

err <- errors(x)

e <- paste(format(err[1:min(5, length(err))]), collapse=" ")

if (length(err) > 5)

e <- paste(e, "...")

cat("Errors: ", e, "\n", sep = "")

# drop attribute and class, then call the next method

class(x) <- setdiff(class(x), "errors")

attr(x, "errors") <- NULL

NextMethod()

}

Dropping the class and the attribute is not needed in general, but in this case we do not want the default printing method (print.default) to show them, so we remove them before calling NextMethod. The latter is not required either, but it is highly advisable if we want to support inheritance, as we will see later.

Now, if we try to print our object again,

x

## Errors: 0.1 0.2 0.3 0.4 0.5 ...

## [1] 1 2 3 4 5 6 7 8 9 10

The output looks nicer. Basically, when print is called, it takes the class, errors, and looks for a method called print.errors. Before we defined it, nothing was found and thus the default method was called. If you call methods(print) you will see that there are hundreds of print methods defined for hundreds of different classes, and our new method is of course listed there.

The next step would be to implement common functions to work with vectors in R, such as [ and c.

`[.errors` <- function(x, ...)

set\_errors(NextMethod(), errors(x)[...])

c.errors <- function(..., recursive = FALSE)

set\_errors(NextMethod(), c(unlist(sapply(list(...), errors))))

Now, we can subset and concatenate errors objects, and their errors are also handled appropriately.

c(x[3], x[7:9])

## Errors: 0.3 0.7 0.8 0.9

## [1] 3 7 8 9

Next, we would like to be able to operate normally with these vectors so that errors are automatically recomputed and propagated. At this point, you are probably guessing that we basically need to start reimplementing a bunch of mathematical functions and arithmetic operations to work with our new class. And you are actually right, but there is a shortcut for that.

Fortunately, R defines what is called *S3 group generics* (see help(groupGeneric)). These are four pre-specified groups of functions containing the following generics:

* Group Math includes mathematical functions:
  + abs, sign, sqrt
  + Rounding functions (floor, ceiling…)
  + Exponential, logarithms and trigonometric functions (cos, sin…)
  + Special functions such as gamma
  + Cumulative sum, product, max and min
* Group Ops includes arithmetic operations:
  + Addition, subtraction, multiplication, power, division and modulo
  + Logic and boolean operators
* Group Summary includes:
  + all, any, sum, prod, min, max, range
* Group Complex includes functions to work with complex numbers:
  + Argument, conjugate, real/imaginary parts, modulus

The existence of these groups means that we can simply write a single method for a whole category to provide for all the functions included. And this is what the errors package does by implementing Math.errors, Ops.errors and Summary.errors. We can take a glimpse into one of these methods:

errors:::Ops.errors

## function (e1, e2)

## {

## if (.Generic %in% c("&", "|", "!", "==", "!=", "<", ">",

## "<=", ">=")) {

## warn\_once("boolean operators not defined for 'errors' objects, uncertainty dropped",

## fun = .Generic, type = "bool")

## return(NextMethod())

## }

## if (!missing(e2)) {

## coercion <- cond2int(!inherits(e1, "errors"), !inherits(e2,

## "errors"))

## if (coercion) {

## warn\_once("non-'errors' operand automatically coerced to an 'errors' object with no uncertainty",

## fun = "Ops", type = "coercion")

## switch(coercion, e1 <- set\_errors(e1), e2 <- set\_errors(e2))

## }

## }

## deriv <- switch(.Generic, `+` = , `-` = if (missing(e2)) {

## e2 <- NA

## list(do.call(.Generic, list(1)), NA)

## } else list(1, do.call(.Generic, list(1))), `\*` = list(.v(e2),

## .v(e1)), `/` = , `%%` = , `%/%` = list(1/.v(e2), -.v(e1)/.v(e2)^2),

## `^` = list(.v(e1)^(.v(e2) - 1) \* .v(e2), .v(e1)^.v(e2) \*

## log(abs(.v(e1)))))

## propagate(unclass(NextMethod()), e1, e2, deriv[[1]], deriv[[2]])

## }

##

##

All the arithmetic operators, as well as logical and boolean operators, share a lot of properties. But still, it is clear that we need some mechanism to be able to do different things depending on which one was called. To solve this, the S3 dispatching mechanism sets for us special variables:

* .Generic contains the name of the generic function.
* .Method contains the name of the method.
* .Class contains the class(es) of the object.

This way, checking .Generic allows us to issue a warning if a non-supported method is called, and then delegate to NextMethod.

Note also that the Ops group is pretty special for two reasons: first, there are unary operators (e.g., -1), so sometimes the second argument is missing, and we need to take that into account; secondly, these operators are commutative, and therefore S3 supports *double dispatch* in this case.

Finally, Ops.errors addresses all the arithmetic operators in a unified way: it computes two derivatives depending on .Generic and then propagates the uncertainty using an auxiliary function that implements the Taylor Series Method.

What we have seen here is just a taste of the actual implementation of the errors package. Now, let’s clear the workspace and load it to check the complete functionality.

rm(list=ls())

library(errors)

set.seed(42)

x <- 1:5 + rnorm(5, sd = 0.01)

y <- 1:5 + rnorm(5, sd = 0.02)

errors(x) <- 0.01

errors(y) <- 0.02

x; y

## Errors: 0.01 0.01 0.01 0.01 0.01

## [1] 1.013710 1.994353 3.003631 4.006329 5.004043

## Errors: 0.02 0.02 0.02 0.02 0.02

## [1] 0.9978775 2.0302304 2.9981068 4.0403685 4.9987457

(z <- x / y)

## Errors: 0.022693105 0.010858436 0.007469263 0.005497048 0.004477051

## [1] 1.0158657 0.9823284 1.0018427 0.9915751 1.0010597

correl(x, x) # one, cannot be changed

## [1] 1 1 1 1 1

correl(x, y) # NULL, not defined yet

## NULL

correl(x, y) <- runif(length(x), -1, 1)

correl(x, y)

## [1] 0.8080628 -0.7225797 0.9777835 0.8933365 -0.8351249

covar(x, y)

## [1] 0.0001616126 -0.0001445159 0.0001955567 0.0001786673 -0.0001670250

(z\_correl <- x / y)

## Errors: 0.013609754 0.013667062 0.003492530 0.002917634 0.005781596

## [1] 1.0158657 0.9823284 1.0018427 0.9915751 1.0010597

(df <- data.frame(x, 3\*x, x^2, sin(x), cumsum(x)))

## Warning: In 'Ops' : non-'errors' operand automatically coerced to an

## 'errors' object with no uncertainty

## x X3...x x.2 sin.x. cumsum.x.

## 1 1.01(1) 3.04(3) 1.03(2) 0.849(5) 1.01(1)

## 2 1.99(1) 5.98(3) 3.98(4) 0.912(4) 3.01(1)

## 3 3.00(1) 9.01(3) 9.02(6) 0.14(1) 6.01(2)

## 4 4.01(1) 12.02(3) 16.05(8) -0.761(6) 10.02(2)

## 5 5.00(1) 15.01(3) 25.0(1) -0.958(3) 15.02(2)

format(df, notation="plus-minus")

## x X3...x x.2 sin.x. cumsum.x.

## 1 1.01 ± 0.01 3.04 ± 0.03 1.03 ± 0.02 0.849 ± 0.005 1.01 ± 0.01

## 2 1.99 ± 0.01 5.98 ± 0.03 3.98 ± 0.04 0.912 ± 0.004 3.01 ± 0.01

## 3 3.00 ± 0.01 9.01 ± 0.03 9.02 ± 0.06 0.14 ± 0.01 6.01 ± 0.02

## 4 4.01 ± 0.01 12.02 ± 0.03 16.05 ± 0.08 -0.761 ± 0.006 10.02 ± 0.02

## 5 5.00 ± 0.01 15.01 ± 0.03 25.0 ± 0.1 -0.958 ± 0.003 15.02 ± 0.02

And much more. This is the complete list of methods implemented in the package:

methods(class="errors")

## [1] [ [[ [[<- [<- all.equal

## [6] as.data.frame as.list as.matrix c cbind

## [11] correl correl<- covar covar<- diff

## [16] drop\_errors errors\_max errors\_min errors errors<-

## [21] format log10 log2 Math mean

## [26] median Ops plot print quantile

## [31] rbind rep set\_correl set\_covar set\_errors

## [36] summary Summary t weighted.mean

## see '?methods' for accessing help and source code

**Implementing new functionality: new generics**

So far, we have explored S3 programming mostly by extending existing generics with methods for a new class of our own. But what if we want to add new functionality? You have probably already guessed that you need to implement your own generics. Let’s take a step back and use a simpler classic example to demonstrate this.

# constructors

circle <- function(r) structure(list(r=r), class="circle")

rectangle <- function(a, b) structure(list(a=a, b=b), class="rectangle")

# generics

perimeter <- function(shape) UseMethod("perimeter")

area <- function(shape) UseMethod("area")

# methods

print.circle <- function(x, ...) with(x, cat("r =", r, "\n"))

perimeter.circle <- function(shape) with(shape, 2 \* pi \* r)

area.circle <- function(shape) with(shape, pi \* r^2)

print.rectangle <- function(x, ...) with(x, cat("a =", a, ", b =", b, "\n"))

perimeter.rectangle <- function(shape) with(shape, 2 \* (a + b))

area.rectangle <- function(shape) with(shape, a \* b)

# usage example

(x <- circle(5))

## r = 5

(y <- rectangle(10, 5))

## a = 10 , b = 5

perimeter(x)

## [1] 31.41593

perimeter(y)

## [1] 30

area(x)

## [1] 78.53982

area(y)

## [1] 50

As you can see, implementing a new generic is as easy as defining a function that calls UseMethod with the name of the generic. More advanced uses may, for example, manipulate the input variables before calling UseMethod, but this basic template would fit most use cases.

**Combining functionality: inheritance**

Inheritance is also possible with S3 programming. Instead of assigning a single class, there may be multiple classes defined as a character vector. When this happens, R dispatches the first class, and subsequent calls to NextMethod look for other methods in the class vector. This means that, if we want our object to resemble parent-child relationships, parent classes must go first in the class vector.

# constructor

shape <- function(name, ..., color) {

shape <- do.call(name, list(...))

shape$color <- color

structure(shape, class=c("shape", class(shape)))

}

# methods

print.shape <- function(x, ...) {

cat(x$color, .Class[2], "\n")

cat("parameters: ")

NextMethod() # call that particular shape's print method

}

# usage example

(x <- shape("circle", 5, color="red"))

## red circle

## parameters: r = 5

(y <- shape("rectangle", 10, 5, color="blue"))

## blue rectangle

## parameters: a = 10 , b = 5

class(x)

## [1] "shape" "circle"

class(y)

## [1] "shape" "rectangle"

perimeter(x)

## [1] 31.41593

perimeter(y)

## [1] 30

area(x)

## [1] 78.53982

area(y)

## [1] 50

This is exactly what the package quantities does to combine the functionality of packages errors and units. As we have seen, errors defines uncertainty metadata for R vectors, and units does the same for measurement units. To achieve a complete calculus system, quantities prepends a superclass to be able to orchestrate units and errors while keeping them completely independent.

library(quantities)

## Loading required package: units

## udunits system database from /usr/share/udunits

# start with a units object and add errors

(x <- set\_units(1:5, "m"))

## Units: [m]

## [1] 1 2 3 4 5

class(x)

## [1] "units"

(x <- set\_errors(x, 0.1))

## Units: [m]

## Errors: 0.1 0.1 0.1 0.1 0.1

## [1] 1 2 3 4 5

class(x)

## [1] "quantities" "units" "errors"

# start with an errors object and add units

(x <- set\_errors(1:5, 0.1))

## Errors: 0.1 0.1 0.1 0.1 0.1

## [1] 1 2 3 4 5

class(x)

## [1] "errors"

(x <- set\_units(x, "m"))

## Units: [m]

## Errors: 0.1 0.1 0.1 0.1 0.1

## [1] 1 2 3 4 5

class(x)

## [1] "quantities" "units" "errors"

# both at the same time

(y <- set\_quantities(1:5, "s", 0.1))

## Units: [s]

## Errors: 0.1 0.1 0.1 0.1 0.1

## [1] 1 2 3 4 5

class(x)

## [1] "quantities" "units" "errors"

# and everything just works

(z <- x / (y\*y))

## Units: [m/s^2]

## Errors: 0.223606798 0.055901699 0.024845200 0.013975425 0.008944272

## [1] 1.0000000 0.5000000 0.3333333 0.2500000 0.2000000

correl(x, z)

## [1] 0.4472136 0.4472136 0.4472136 0.4472136 0.4472136

sum(z)

## 2.3(2) [m/s^2]

**Summary**

S3 programming requires three steps:

* Define a constructor (possibly a validator too) that sets the class and the attributes.
* Define new generics with UseMethod.
* Implement methods for new and existing generics with a proper use of NextMethod to support inheritance and also avoid code duplication.

In contrast to other OO systems, objects hold the data, and methods live outside as independent functions. The developer needs to invest more effort and care to ensure correctness and support inheritance, but in exchange, it is a very flexible system, because it is easy to extend without modifying the object.