SimpleS3

by Nicholas Tierney

Abstract Writing functions in R is an important skill for anyone using R. S3 methods allow for functions to be generalised across different classes and are easy to implement. Whilst many R users are be adept at creating their own functions, it seems that there is room for many more to take advantage of R's S3 methods. This paper provides a simple and targeted guide to explain what S3 methods are, why people should them, and how they can do it.

Note

This file is only a basic article template. For full details of *The R Journal* style and information on how to prepare your article for submission, see the Instructions for Authors.

Introduction

A standard principle of programming is DRY - Don't Repeat Yourself [ref]. Under this axiom, the copying and pasting of the same or similar code (copypasta), is avoided and replaced with a function. Having one function to replace several of the same or similar coded sections simplifies code maintenance as it means that only one section of code needs to be maintained, instead of several. This means that if the code breaks, then one simply needs to update the function, rather than finding all of the coded sections that are now broken.

S3 methods in the R programming language are a way of writing functions in R that do different things for objects of different classes. S3 methods are so named as the methods shipped with the release of the third version of the "S" programming language, which R was heavily based upon [reference]. Hence, methods for S 3.0 = S3 Methods.

The function `summary()` is an S3 method. When applied to an object of class `data.frame`, `summary` shows descriptive statistics (Mean, SD, etc.) for each variable. For example, `iris` is of class `data.frame`:

```
class(iris)
```

```
#> [1] "data.frame"
```

So applying `summary` to `iris` gives us summary information relevant to a dataframe

```
summary(iris)
```

```
#>   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
#>  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
#>  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
#>  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
#>  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
#>  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
#>  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
#>        Species  
#>  setosa    :50  
#>  versicolor:50  
#>  virginica :50  
#> 
#> 
#> 
```

`summary` also performs differently when applied to different object. In fact, you can find all the classes that work with an S3 method by typing the following:

```
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.PDF_Dictionary*
#> [21] summary.PDF_Stream*           summary.POSIXct
#> [23] summary.POSIXlt               summary.ppr*
#> [25] summary.prcomp*               summary.princomp*
#> [27] summary.proc_time             summary.srcfile
#> [29] summary.srcref                summary.stepfun
#> [31] summary.stl*                  summary.table
#> [33] summary.tukeysmooth*
#> see '?methods' for accessing help and source code
```

There's over 30 different methods!

We can use summary on a linear model, for example:

```
lm_iris <- lm(Sepal.Length ~ Sepal.Width, data = iris)

summary(lm_iris)
```

```
#>
#> Call:
#> lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
#>
#> Residuals:
#>     Min      1Q  Median      3Q     Max
#> -1.5561 -0.6333 -0.1120  0.5579  2.2226
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)   6.5262     0.4789   13.63   <2e-16 ***
#> Sepal.Width  -0.2234     0.1551   -1.44    0.152
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.8251 on 148 degrees of freedom
#> Multiple R-squared:  0.01382,    Adjusted R-squared:  0.007159
#> F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

`summary` produces a description of the linear model, describing how it was called (`call`), as well as the `residuals`, `coefficients`, `t-values`, `p-values`, R^2, and more. This output is **completely** different to the information output from `summary` used for the `iris` dataframe.

So how does the same function, `summary` perform differently for different objects? The answer is that R is helpful, and *hides* this information. There are in fact, many different `summary` functions. For example:

- `summary.lm`
- `summary.data.frame`
- `summary.Date`
- `summary.matrix`

Being an S3 method, `summary` calls the appropriate function based upon the class of the object it operates on. So using `summary` on an object of class "Date" will evoke the function, `summary.Date`. **But all you need to do is type `summary`**, and the S3 method does the rest. By abstracting away this detail (the object class), the intent becomes clearer.

To further illustrate, using `summary` on the `iris` data will actually call the function `summary.data.frame`, since `iris` is of class `data.frame`. We can find the class of an object using `class`

```
class(iris)
```

```
#> [1] "data.frame"
```

```
summary.data.frame(iris)
```

```
#>   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
#>  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
#>  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
#>  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
#>  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
#>  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
#>  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
#>        Species  
#>  setosa    :50  
#>  versicolor:50  
#>  virginica :50  
#>                 
#>                 
#>                 
```

which is the same as summary(iris)

```
sum1_df <- summary.data.frame(iris)

sum2_df <- summary(iris)

all.equal(sum1_df, sum2_df)
```

```
#> [1] TRUE
```

And using summary on the linear model object, `lm_iris` performs:

```
summary.lm(lm_iris)
```

```
#> 
#> Call:
#> lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
#> 
#> Residuals:
#>     Min      1Q  Median      3Q     Max 
#> -1.5561 -0.6333 -0.1120  0.5579  2.2226 
#> 
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)    
#> (Intercept)   6.5262     0.4789   13.63   <2e-16 ***
#> Sepal.Width  -0.2234     0.1551   -1.44    0.152    
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#> 
#> Residual standard error: 0.8251 on 148 degrees of freedom
#> Multiple R-squared:  0.01382,    Adjusted R-squared:  0.007159 
#> F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

the same as summary(lm_iris)

```
sum1_lm <- summary.lm(lm_iris)

sum2_lm <- summary(lm_iris)

all.equal(sum1_lm, sum2_lm)
```

```
#> [1] TRUE
```

One could coerce a different method upon a different class, for example using `summary.data.frame` on an "lm" object:

```
summary.data.frame(lm_iris)
```

```
#>   coefficients       residuals          effects               rank  
#>  Min.   :-0.2234   Min.   :-1.5561   Min.   :-71.56593   Min.   :2  
#>  1st Qu.: 1.4640   1st Qu.:-0.6333   1st Qu.: -0.65192   1st Qu.:2  
#>  Median : 3.1514   Median :-0.1120   Median : -0.00897   Median :2  
#>  Mean   : 3.1514   Mean   : 0.0000   Mean   : -0.42040   Mean   :2
#>  3rd Qu.: 4.8388   3rd Qu.: 0.5579   3rd Qu.:  0.61051   3rd Qu.:2
#>  Max.   : 6.5262   Max.   : 2.2225   Max.   :  2.15225   Max.   :2
#>  fitted.values       assign     qr.Length  qr.Class  qr.Mode  df.residual
#>  Min.   :5.543   Min.   :0.00   300      -none-   numeric    Min.   :148
#>  1st Qu.:5.789   1st Qu.:0.25     2      -none-   numeric    1st Qu.:148
#>  Median :5.856   Median :0.50     2      -none-   numeric    Median :148
#>  Mean   :5.843   Mean   :0.50     1      -none-   numeric    Mean   :148
#>  3rd Qu.:5.901   3rd Qu.:0.75     1      -none-   numeric    3rd Qu.:148
#>  Max.   :6.080   Max.   :1.00                                Max.   :148
#>    xlevels         call         terms
#>  Length:0      Length:3      Length:3
#>  Class :list   Class :call   Class1:terms
#>  Mode  :list   Mode  :call   Class2:formula
#>                              Mode  :call
#>
#>
#>  model.Sepal.Length  model.Sepal.Width
#>  Min.   :4.300000    Min.   :2.000000
#>  1st Qu.:5.100000    1st Qu.:2.800000
#>  Median :5.800000    Median :3.000000
#>  Mean   :5.843333    Mean   :3.057333
#>  3rd Qu.:6.400000    3rd Qu.:3.300000
#>  Max.   :7.900000    Max.   :4.400000
```

However the output may be a bit confusing.

To summarize, the important feature of S3 methods worth noting is that only the **first part**, `summary`, is required to be used on these objects of different classes.

Why hide the text?

Hiding the trailing text after the `.` avoids the need to use a different `summary` function for every class. This means that one does not need to remember to use `summary.lm` for linear models, or `summary.data.frame` for data frames, or `summary.aProposterousClassOfObject`. By using S3 methods, cognitive load is reduced - you don't have to think as much to remember what class an object is - and the commands are more intuitive. To get a summary of most objects, use `summary`, to plot most objects, use `plot`. Perhaps the most nifty feature of all is that a user can create their own S3 methods using the same functions such as `summary` and `plot`. This means a user can create their own special class of object

```
test_class <- 1:10

class(test_class) <- "myclass"

class(test_class)

#> [1] "myclass"
```

and then write their own S3 method for it - e.g., `summary.myclass` or `plot.myclass`, each proiding appropriate summary information, or nice plots, for that object.

How to make your own S3 method?

Creating your own S3 method is not particularly difficult and is usually highly practical. A use case scenario for creating an S3 method is now discussed.

The Residual Sums of Squares (RSS), $\sum(Y_i - \hat{Y})^2$ is a useful metric for determining model accuracy for continuous outcomes. For example, for a Classification and Regression Tree

```
library(rpart)

fit.rpart <- rpart(Sepal.Width ~ Sepal.Length + Petal.Length + Petal.Width + Species, data = iris)
```

The RSS is calculated as

```
print_rss <- sum(residuals(fit.rpart)**2)
```

One might be inclined to write a function to perform this task

```
rss <- function(x){

  sum(residuals(x)**2)

}

rss(fit.rpart)

#> [1] 10.17245
```

However, there are many different decision tree models that one would like to compare, say boosted regression trees (BRT), and random forests (RF). The same code will not work:

```
library(randomForest)

#> randomForest 4.6-12

#> Type rfNews() to see new features/changes/bug fixes.

set.seed(71)
fit.rf <- randomForest(Sepal.Length ~ ., data=iris, importance=TRUE,
                        proximity=TRUE)

rss(fit.rf)

#> [1] 0
```

In this case, one could write three functions, one for each decision tree method: "rss_rpart", "rss_brt", and "rss_rf". But to avoid having three functions and instead use just one, one could place all three functions inside of one function, using an if-then-else clause to direct the object of the appropriate class to the appropriate method. This shall be referred to as a "Poor man's S3 method".

\begin{Schunk} \begin{Sinput} dt_rss <- function (x){

if ("rpart" %in% class(x)) {

result <- sum((residuals(x)**2))

return(result)

}

else if ("gbm" %in% class(x)) {

result <- sum(x$residuals**n2)

return(result)

}

else if ("randomForest" %in% class(x)) {

temp <- x$y - x$predicted

result <- sum(temp**2)

return(result)

}

else warning(paste(class(x), "is not of an rpart, gbm, or randomForest object")) } \end{Sinput} \end{Schunk}

Here it is in action

```
dt_rss(fit.rpart)

#> [1] 10.17245
```

The RSS method works, and if it is applied to a class that is not known, a special message is provided

```
fit.lm <- lm(Sepal.Width ~ Species, data = iris)

dt_rss(fit.lm)
```

```
#> Warning in dt_rss(fit.lm): lm is not of an rpart, gbm, or randomForest
#> object
```

The "poor man's S3 method" does what it needs to do. However, one must ask how sustainable this would be for an entire programming language? Imagine if a colleague creates a new tree method that needs it's own `rss()`. He will need to convince the maintainer to add his class into your ifelse() chain. Failing this, he could just overwrite the function `rss()`, with predictably disastrous results. In reality, it's probably better to do all of these things with one method. R's S3 methods mean that R developers can utilise a common interface without having to update it when new classes come along. It also means overloading clashes are less likely.

So let us create an S3 method to demonstrate.

First define the S3 method with `UseMethod()`

```
rss <- function(x) UseMethod("rss")
```

This creates the building block of an S3 method, the "root", if you will.

Here we have specified that our method will be called `rss`. Now we need to create the special cases of rss - the methods `rss.rpart`, `rss.gbm`, and `rss.randomForest`, where the sections of code after `rss.` are the classes of object we want them to work on.

\begin{Schunk} \begin{Sinput} rss.rpart <- function(x){

sum((residuals(x)**2))

}

rss.gbm <- function(x){

sum(x$residuals**2)

}

rss.randomForest <- function(x){

res <- x$y - x$predicted

sum(res**2)

} \end{Sinput} \end{Schunk}

A default method can also be created - `rss.default` - which, as the name suggests, is the default method when the argument `x` is not a class that has a specific version of the method defined.

```
rss.default <- function(x, ...){

  warning(paste("RSS does not know how to handle object of class ",
                class(x),
                "and can only be used on classes rpart, gbm, and randomForest"))

        }
```

In this case a warning is issued, to let the user know that the object class they were using was not appropriate.

We can now apply the `rss` method to an `rpart` model

```
rss(fit.rpart)
```

```
#> [1] 10.17245
```

Also observe what happens when the object used is not of the decision tree classes

```
rss(lm.fit)
```

```
#> Warning in rss.default(lm.fit): RSS does not know how to handle object of
#> class function and can only be used on classes rpart, gbm, and randomForest
```

This guide to S3 methods was written to provide R users with the minimal amount of information to start building their own S3 methods. For a more complete treatment on S3 methods, see Advanced-R, R Packages, and this blog, this resource.

Extras

For the uninitiated, you may find the class of an object using the command, `class()`, on the object. For example:

```
x <- c(1, 2, 3, 4, 5)
```

```
x
```

```
#> [1] 1 2 3 4 5
```

```
class(x)
```

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

Here, showing that the object `x` is of class `numeric`.

`str()` can also provide more information:

```
str(x)
```

```
#>  num [1:5] 1 2 3 4 5
```

In this case, revealing that `x` is `numeric`, showing its contents.

> Miles: This raises an interesting question for me about S3 methods.. When functions take multiple arguments.. like plot(), how do you nominate which argument is used to switch the function body based on type? Is it just the first one? Furthermore, do all S3 methods need to have identical arguments? Can lazy dots be used? it looks like they can. SHOULD they be used?

Idea

Idea: Make a cheatsheet / infographic for writing functions in R, and for making s3 methods. Let's call it "The Anatomy of S3 Methods"

Nicholas Tierney
Queensland University of Technology
Level 8, Y Block, Main Drive, QUT, Brisbane, Australia
nicholas.tierney@gmail.com