package: s3x 0.3.0

Enhanced S3 Programming (Rough Draft)

Charlotte Maia

October 25, 2011

This vignette provides an overview of the s3x system, for enhanced S3 programming, including mixing object oriented programming with numerical programming.

The S3 System

Class-based programming is necessary to implement class-based models effectively, which in turn, is necessary for creating real-world models.

The view of the author is that R's S3 system, supports a weak form of class-based programming, which is a good thing.

In many "other" object oriented programming systems (e.g. C++, Java and S4), classes are definitions of objects. We have to define (explicit) classes, in order to create objects. In S3, classes are descriptions of objects. We can create objects with (implicit or explicit) classes. After creating objects, we can optionally change their classes.

Either way, if we want to create new data-types, we must first define new classes. Many systems have a special syntax for defining classes, however S3 uses standard functions as constructors.

Many systems support both class-level and object-level methods, hence we can call either:

```
my_class (my_args)
```

```
my_object.my_method (my_args)
```

S3 supports generic functions and class-level methods, hence we can call either:

```
my_generic (my_object, my_args)
```

```
my_method.my_class (my_object, my_args)
```

In theory, this system could be extended for multiple despatch, hence we could call either:

```
my_generic (my_object_1, my_object_2, my_args)
```

```
my_method.my_class_1.my_class_2 (my_object_1, my_object_2, my_args)
```

One small problem occurs for mutator methods, where (in general) we must duplicate the object:

```
my_object = my_generic (my_object, my_args)
```

Unfortunately, S3 allows generic functions to have argument names, which restricts method argument names.

In many systems, classes define (and restrict) an object's attributes. After creating an object, we can't change what attributes it has, only it's attributes' values. In S3, classes don't define (and don't restrict) an object's attributes. After creating an object, we can change what attributes it has.

Other major strengths of R's S3 system include:

1. All data-types (including lists, R expressions, functions, matrices and vectors) are first class objects, hence functions can take R expressions or other functions as arguments.
2. All objects can have attributes.
3. Very strong support for lists.
4. Relatively strong support for numerical programming.
   Noting that specialised tasks may require foreign language implementations.

Other major weaknesses with R's S3 system include:

1. A lack of object references.
2. A lack of clean exception handling.
3. A lack of a member (or attribute) operator.

Enhanced S3 Programming

The s3x system is built on top of R's S3 system.

This package represents the culmination of several attempts to enhance R's object oriented features.

Initially, the author was interested in creating a C++-like or Java-like system. Then later, she became more interested in mixing object oriented programming with functions.

Then more recently, she became interested in a two-pronged approach. Firstly, mixing object-oriented programming with lists, which forms the basis for general purpose object oriented programming. Secondly, mixing object oriented programming with mathematical primitives, which forms the basis for basis for mixing object oriented programming with numerical programming.

Currently, the package has five over-arching goals:

1. To support highly readable R source code.
2. To support mainstream object oriented programming.
3. To support standard R lists and list-like syntax.
4. To provide enhanced primitives,
   for mixing object oriented programming with numerical programming.
5. To provide utilities,
   for mixing object oriented programming with numerical programming.

The author uses the term "enhanced primitives", to describe top-level classes that mix object oriented features with major mathematical features. Currently, enhanced primitives

include:

1. Enhanced tables (TABLE objects), which are similar to data.frame objects.
2. Enhanced functions (FUNCTION objects).
3. Enhanced vectors (VECTOR objects).

The main difference between standard R objects and enhanced primitives, is that enhanced primitives use an alternative attribute system. The author uses the term "R attributes" to describe standard R attributes and "object attributes" to describe named list elements and enhanced primitive attributes.

Like lists, enhanced primitives, use the "$" operator as an object attribute operator, to set and get their object attributes. Enhanced functions support self-referencing, where ".$" maybe used within their bodies to get object attribute values. This is particularly useful for interpolated functions.

Note that with the exception of lists, object attributes are stored as a standard R attribute named ".".

Another difference between standard R objects and enhanced primitives is in subsetting. By default, for enhanced tables and enhanced vectors, single brackets, return a "sub" object of the same class with the same object attributes, where double brackets (which can have vectorised arguments) return components standard R vectors.

In addition to enhanced primitives, the package provides the following features:

1. General purpose object oriented features, including:
   (a) Constructor utilities.
   (b) Generic functions (redefinitions), with no named arguments.
   (c) Object references.
2. Tabular utilities (early prototypes).
3. Functional utilities, including:
   (a) Smoothing (locally-weighted least squares).
   (b) Linear interpolation.
4. Other utilities, including:
   (a) Formatted vectors.
   (b) Generic and nonrandom sampling.

Other features that are being considered for the future, include exception handling, applications programming tools (e.g. text menus), an enhanced primitive superclass, more enhanced primitives (graphs, trees, matrices and equations), 3d plots (possibly just wrappers), more tabular utilities, more smoothing, optimisation, random variables, multiple despatch and despatch for object references.

General Purpose Object Oriented Features

Constructors

The s3x package provides two utility functions extend and implant to simplify construction.

The extend function takes an object and a subclass name, then returns the extended object. If the object has an implicit class, then it's class is set to the new class. If it an explicit class, then it's class is concatenated. The subclass name can be vectorised, however the order is the opposite to S3, superclass first, subclass second.

```
R> object = list ()
R> extend (object, "myclass")
```

```
list()
attr(,"class")
[1] "myclass"
```

The implant function takes an object (with a "$<-" method) and a list of object attributes, which are added to the object. The list can't include "...". The arguments can be named or unnamed. If unnamed, they default to the corresponding identifier.

```
R> x = 10
R> y = 20
R> implant (list (), x, y, z=x + y)
```

```
$x
[1] 10

$y
[1] 20

$z
[1] 30
```

The extend function, can also except attributes, however the author recommends against this except for one line constructors.

```
R> myclass = function (x, y)
           extend (list (), "myclass", x, y, z=x + y)
R> myobject = myclass (1, 2)
R> myobject
```

```
$x
[1] 1

$y
[1] 2

$z
[1] 3

attr(,"class")
[1] "myclass"
```

Generic Functions

The current version of this package, redefines all generics from the base and graphics packages. They simply call the standard versions. This may be modified in future versions.

```
R> print

function (...)
base::"print"(...)
<environment: namespace:s3x>
```

After loading this package, we can write methods (say print.myclass) that don't require the argument to be named "x".

Object Referencing

Environments can be used to support object referencing, however this produces slightly verbose and confusing syntax.

The package provides objref objects to emulate traditional object references.

An object reference is created via the objref function and dereferenced via the deref function.

```
R> ref = objref (1:3)
R> ref

objref -> integer

R> deref (ref)

[1] 1 2 3
```

Currently, there are some limitations and performance issues with these objects, which will hopefully be fixed in the near future.

The main limitation relates to method despatch, which should (however isn't) be determined by the class of the referenced object. Another limitation, is that there's no direct support for changing what a reference, references.

A few methods are defined, such $ and bracket operators, which apply to the referenced object.

The following example illustrates typical reference behaviour, where changes to a referenced object are reflected in another reference

```
R> duplicate = ref
R> ref [1] = 0
R> deref (duplicate)

[1] 0 2 3
```

Enhanced Vectors

Enhanced vectors (VECTOR objects) are the simplest of all enhanced primitives.

We can create an enhanced vector and assign attributes to it, which is particularly useful for representing measurements along with units.

Enhanced vectors are created from standard vectors (or any object that can be coerced to a standard vector).

Simple example based on the cars dataset (which is imperial).

```
R> #a speed object
R> speed = VECTOR (cars$speed)
R> speed$unit = "mph"
R> speed
```

```
VECTOR
 [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14
[23] 14 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22
[45] 23 24 24 24 24 25
object attributes:
unit
```

```
R> speed$unit
```

```
[1] "mph"
```

```
R> #a distance object
R> distance = VECTOR (cars$dist)
R> distance$unit = "ft"
R> distance
```

```
VECTOR
 [1]   2  10   4  22  16  10  18  26  34  17  28  14  20  24  28  26  34
[18]  34  46  26  36  60  80  20  26  54  32  40  32  40  50  42  56  76
[35]  84  36  46  68  32  48  52  56  64  66  54  70  92  93 120  85
object attributes:
unit
```

```
R> distance$unit
```

```
[1] "ft"
```

A subtle difference between standard vectors and enhanced vectors is that subsetting via single brackets returns an object with a copy of the original object's attributes and that subsetting via double brackets returns a standard vector (including potentially more than one element).

```
R> distance [1:10]
```

```
VECTOR
 [1]  2 10  4 22 16 10 18 26 34 17
object attributes:
unit
```

```
R> distance [[1:10]]
```

```
 [1]  2 10  4 22 16 10 18 26 34 17
```

Note that when we perform arithmetic operations that involve two enhanced vectors or an enhanced vector an another object, in general, the resulting object will have the first object's class and attributes.

Enhanced Tables

Enhanced tables (TABLE objects) are early prototypes and should be used cautiously.

Currently, enhanced tables are created from lists (or list subclasses) and offer little data validation.

Like data.frames, enhanced tables extend standard lists, with each list element corresponding to a column. However unlike data.frames, by default attributes aren't stripped and the $ operator accesses object attributes rather than list elements.

```
R> #table with attributes
R> table = TABLE (cars)
R> table$unit_speed = "mph"
R> table$unit_distance = "ft"
R> table [1:10,]

   speed dist
1      4    2
2      4   10
3      7    4
4      7   22
5      8   16
6      9   10
7     10   18
8     10   26
9     10   34
10    11   17

R> table$unit_speed

[1] "mph"

R> #like enhanced vectors, double brackets are used to access the elements
R> table [["speed"]]

 [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14
[23] 14 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22
[45] 23 24 24 24 24 25

R> #or
R> table [[,"speed"]]

 [1]  4  4  7  7  8  9 10 10 10 11 11 12 12 12 12 13 13 13 13 14 14 14
[23] 14 15 15 15 16 16 17 17 17 18 18 18 18 19 19 19 20 20 20 20 20 22
[45] 23 24 24 24 24 25
```

We can access all attributes, use the objattr function.

```
R> objattr (table)

$unit_speed
[1] "mph"


$unit_distance
[1] "ft"
```

Utility functions are still being designed:

```
R> summary (table)
```

```
TABLE
  variable   class NAs  mean min max
1    speed numeric   0  15.4   4  25
2     dist numeric   0 42.98   2 120
```

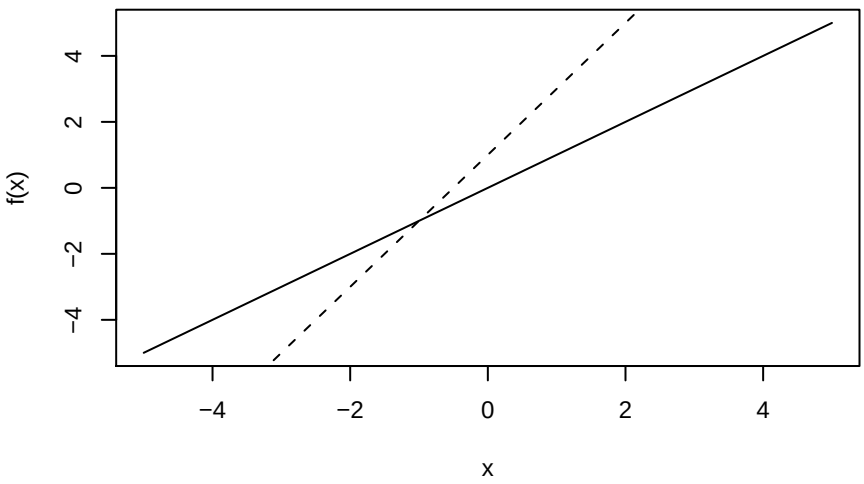
Enhanced Functions

Enhanced functions (FUNCTION objects) provide the same extension for attributes, except that they also support self-referencing (read only). Currently, there's a bug in the print method.

```
R> straight_line = function (intercept, slope)
   {        f_seed = function (x) .$intercept + .$slope * x
           f = FUNCTION (f_seed)
           f$intercept = intercept
           f$slope = slope
           f
   }
R> f1 = straight_line (0, 1)
R> f2 = straight_line (1, 2)
R> f1

FUNCTION (x)
{  .$intercept + .$slope * x
}
object attributes:
intercept, slope

R> objattr (f1)

$intercept
[1] 0


$slope
[1] 1

R> f1 (1:4)

[1] 1 2 3 4

R> f2 (1:4)

[1] 3 5 7 9
```

It's possible to specify an xlim argument in the enhanced function constructor. If we don't and we want to plot our functions, then we must include it in the plot functions.

```
R> plot (f1, xlim=c (-5, 5) )
R> lines (f2, lty=2, xlim=c (-5, 5) )
```



Note that if a function f calls a function g, g doesn't automatically have access to f's attributes. The object . is a named list of attributes, which needs to be given to g as an argument, for g to use those attributes.

Tabular Utilities

The function, read_package_data, can be used to create an enhanced table from a package dataset. Another function, read_data_file can be used read other data files. These functions call read.table, however have different defaults. Refer to the man file for more info.

Note that the print method for table, converts the table to a data.frame. This may be changed in future versions.

```
R> table = read_package_data ("rrv", "markowitz.csv")
R> print (table, 1, row.names=FALSE)
```

```
 Year Am.T. A.T. & T. U.S.S. G.M. A.T. & Sfe C.C. Bdn. Frstn. S.S.
 1937  -0.3      -0.2   -0.3 -0.5       -0.5 -0.1 -0.3   -0.4 -0.4
 1938   0.5       0.1    0.3  0.7        0.1  0.2  0.1    0.3  0.2
 1939   0.1       0.2    0.0  0.2       -0.4 -0.1  0.3   -0.1 -0.3
 1940  -0.1       0.0    0.1  0.0       -0.2 -0.1 -0.1   -0.1  0.0
 1941  -0.3      -0.2   -0.2 -0.3        0.6 -0.2  0.1   -0.2 -0.2
 1942   0.0       0.1    0.0  0.5        0.9  0.2  0.3    1.1  0.1
 1943   0.4       0.3    0.1  0.3        0.3  0.4  0.3    0.6  0.6
 1944   0.2       0.1    0.3  0.3        0.6  0.2  0.2    0.5  0.3
 1945   0.4       0.2    0.4  0.2        0.4  0.3  0.4    0.2  0.6
 1946  -0.1       0.0   -0.1 -0.3        0.0 -0.2  0.2   -0.1  0.3
 1947  -0.1      -0.1    0.2  0.1        0.0  0.4 -0.1    0.0  0.2
 1948   0.0       0.1    0.0  0.1        0.2 -0.2  0.0    0.0  0.1
 1949   0.3       0.0    0.1  0.3        0.1  0.2  0.3    0.2 -0.2
 1950  -0.1       0.1    0.7  0.3        0.6 -0.2  0.1    0.6  0.3
 1951   0.0       0.1    0.0  0.2        0.0 -0.1  0.1   -0.1  0.3
 1952   0.1       0.1    0.1  0.4        0.4  0.1  0.1    0.2  0.1
 1953   0.0       0.0    0.0 -0.1        0.0  0.1  0.2   -0.1  0.0
 1954   0.2       0.2    0.9  0.7        0.5  0.1  0.1    0.8  0.2
```

Functional Utilities

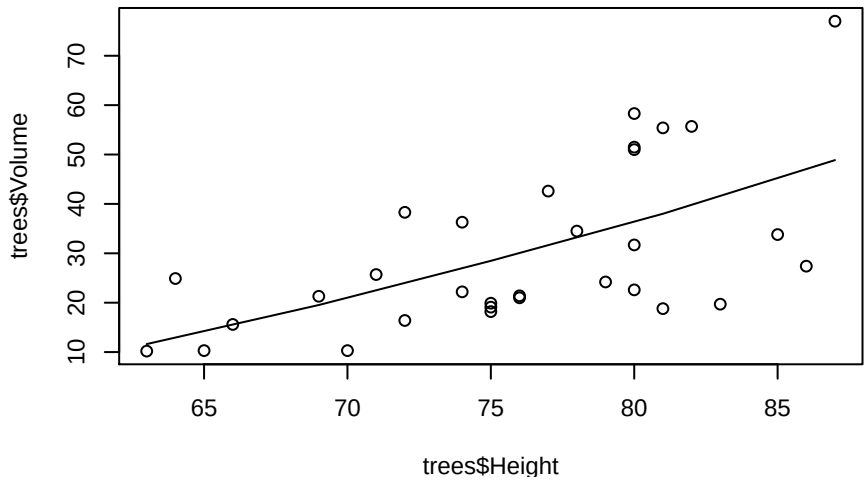
Smoothing

The function lps, fits local polynomials, via locally-weighted least squares.

The function lps_series, is a wrapper, that takes x and y values and returns a series of length n.

By default, the polynomials are quadratic and have a smoothness parameter (relative bandwidth) equal to one (actual bandwidth = smoothness * diff (range (x))).

```
R> m = lps_series (trees$Height, trees$Volume, degree=1, smoothness=2, n=5)
R> plot (trees$Height, trees$Volume)
R> lines (m)
```




```
R> m
```

```
      u        v
[1,] 63 11.62343
[2,] 69 19.52792
[3,] 75 28.47463
[4,] 81 38.00443
[5,] 87 48.86720
```

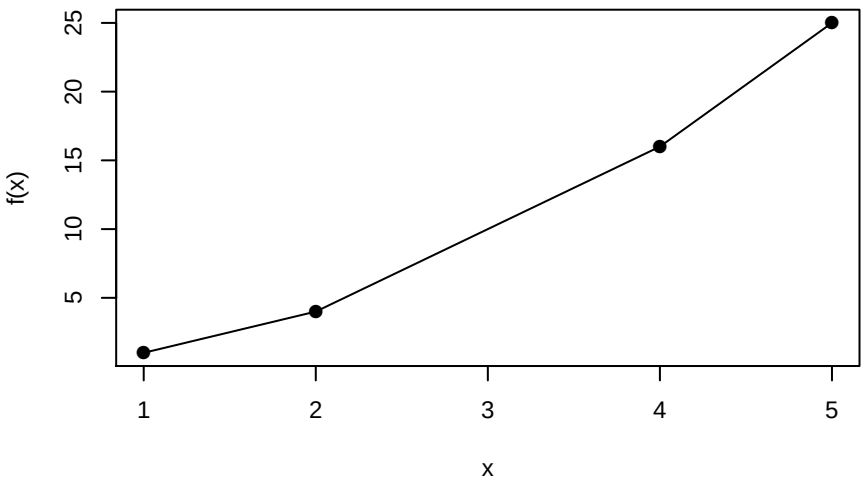
Interpolation

Univariate linear interpolation (for both regularly spaced and irregularly spaced series) can be achieved via the interpolate function. The function takes an x vector and a y vector, with the restriction that the x vector is distinct sorted values, in ascending order, then given a new set of x values (sometimes denoted u), the interpolated values v are computed.

```
R> x = c (1:2, 4:5)
R> y = x^2
R> f = interpolate (x, y)
```

We can plot the function, along with the series.

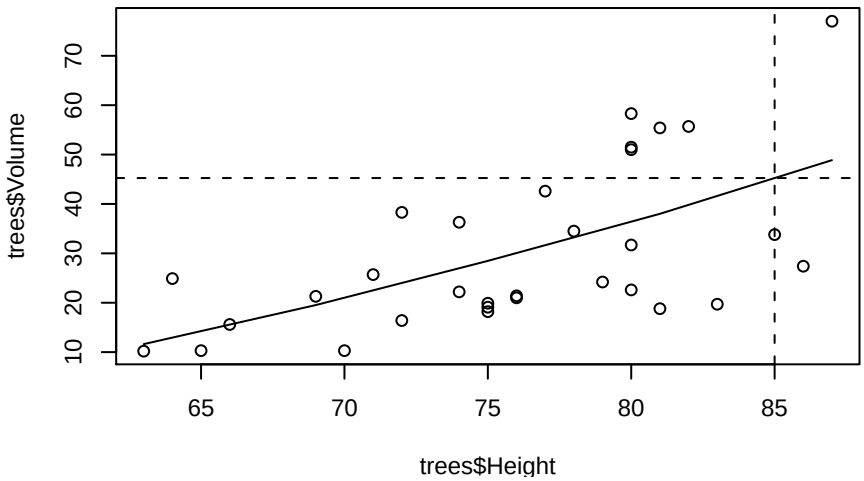
```
R> plot (f, points=TRUE)
```



Mixed Smoothing-Interpolation

The lps_interpolate function is the same as the lps_series function, except that it returns an interpolated function. Evaluating the function, yields fitted values.

```
R> f = lps_interpolate (trees$Height, trees$Volume, degree=1, smoothness=2, n=5)
R> plot (trees$Height, trees$Volume)
R> lines (f)
R> abline (v=85, h=f (85), lty=2)
```



Other Utilities

Formatted Vectors

The vector_style class, can be used to format integer and numeric vectors, it provides commas and potentially currency symbols.

```
R> x = 10e6 * rnorm (10)
R> vector_style (x)
```

```
 [1]  -1,244,211.96 -13,738,101.71  11,669,752.17 -15,399,093.21
 [5]  10,536,011.19   6,144,346.41  24,572,307.87   1,010,955.35
 [9]  -1,602,894.50  34,547,812.26
```

```
R> vector_style (x, 0, currency="$")
```

```
 [1]  -$1,244,212 -$13,738,102  $11,669,752 -$15,399,093  $10,536,011
 [6]   $6,144,346  $24,572,308   $1,010,955  -$1,602,895  $34,547,812
```

Generic Sampling

The sample function is redefined as a generic.

The enhanced table method calls sample_deterministic, which gives the first n rows and last m rows, by default the first three and last three.

```
R> sample (table)
```

```
  Year  Am.T. A.T. & T. U.S.S.   G.M. A.T. & Sfe   C.C.   Bdn. Frstn.
1 1937 -0.305    -0.173 -0.318 -0.477     -0.457 -0.065 -0.319 -0.400
2 1938  0.513     0.098  0.285  0.714      0.107  0.238  0.076  0.336
3 1939  0.055     0.200 -0.047  0.165     -0.424 -0.078  0.318 -0.093
4 1952  0.128     0.083  0.131  0.390      0.434  0.079  0.109  0.175
5 1953 -0.010     0.035  0.006 -0.072     -0.027  0.067  0.210 -0.084
6 1954  0.154     0.176  0.908  0.715      0.469  0.077  0.112  0.756
    S.S.
1 -0.435
2  0.238
3 -0.295
4  0.062
5 -0.048
6  0.185
```

```
R> sample (table, 4)
```

```
  Year  Am.T. A.T. & T. U.S.S.   G.M. A.T. & Sfe   C.C.   Bdn. Frstn.
1 1937 -0.305    -0.173 -0.318 -0.477     -0.457 -0.065 -0.319 -0.400
2 1938  0.513     0.098  0.285  0.714      0.107  0.238  0.076  0.336
3 1939  0.055     0.200 -0.047  0.165     -0.424 -0.078  0.318 -0.093
4 1940 -0.126     0.030  0.104 -0.043     -0.189 -0.077 -0.051 -0.090
5 1951  0.016     0.090  0.021  0.195      0.040 -0.064  0.054 -0.131
6 1952  0.128     0.083  0.131  0.390      0.434  0.079  0.109  0.175
7 1953 -0.010     0.035  0.006 -0.072     -0.027  0.067  0.210 -0.084
8 1954  0.154     0.176  0.908  0.715      0.469  0.077  0.112  0.756
    S.S.
1 -0.435
2  0.238
3 -0.295
4 -0.036
5  0.333
6  0.062
7 -0.048
8  0.185
```

```
R> sample (table, 2, 1)
```

```
  Year  Am.T. A.T. & T. U.S.S.   G.M. A.T. & Sfe   C.C.   Bdn. Frstn.
```

```
  1 1937 -0.305    -0.173 -0.318 -0.477     -0.457 -0.065 -0.319 -0.400
  2 1938  0.513     0.098  0.285  0.714      0.107  0.238  0.076  0.336
  3 1954  0.154     0.176  0.908  0.715      0.469  0.077  0.112  0.756
     S.S.
  1 -0.435
  2  0.238
  3  0.185
```

The default method and a sample_random function, call the standard sample function.