xts: Extensible Time Series

Jeffrey A. Ryan   Joshua M. Ulrich

May 18, 2008

Contents

1 Introduction

The statistical language `R` [5] offers the time-series analyst a variety of mechanisms to both store and manage time-indexed data. Native `R` classes potentially suitable for time-series data include `data.frame`, `matrix`, `vector`, and `ts` objects. Additional time-series tools have been subsequently introduced in contributed packages to handle some of the domain-specific shortcomings of the native `R` classes. These include `irts` from the `tseries` package[2], `timeSeries` from the `Rmetrics` bundle[3], and `zoo` [1] from their respective packages. Each of these contributed classes provides unique solution to many of the issues related to working with time-series in R.

While it seems a bit paradoxical with all the current options available, what `R` really needed was one more time-series class. Why? Users of R have had many choices over the years for managing time-series data. This variety has meant that developers have had to pick and choose the classes they would support, or impose the necessary conversions upon the end-user. With the sheer magnitude of software packages available from CRAN, it has become a challenge for users and developers to select a time-series class that will manage the needs of the individual user, as well as remain compatible with the broadest audience.

What may be sufficient for one use — say a quick correlation matrix may be too limiting when more information needs to be incorporated in a complex calculation. This is especially true for functions that rely on time-based indexes to be manipulated or checked.

The previous solution to managing different data needs often involved a series of `as` calls, to coerce objects from one type to another. While this may be sufficient for many cases, it is less flexible than allowing the users to simply use the object they are accustomed to, or quite possibly require. Additionally, all current coercion methods fail to maintain the original object's data in its entirety. Converting from a `timeSeries` class to `zoo` would cause attributes such as *FinCenter*, *format*, and *recordIDs* to be lost. Converting back to a `timeSeries` would then add new values different than the original. For many calculations that do not modify the data, this is most likely an acceptable side effect. For functions that convert data — such as `xts`'s `to.period` — it limits the value of the function, as the returned object is missing much of what may have been a factor in the original class consideration.

One of the most important additions the new `xts` class makes to the R user's workflow doesn't use `xts` at all, at least not explicitly. By converting data to `xts` inside a function, the function developer is guaranteed to have to only manage a single class of objects. It becomes unecessary to write specific methods to handle different data. While many functions do have methods to accommodate different classes, most do not. Before `xts`, the `chartSeries` function in the `quantmod` package[6] was only able to handle `zoo` objects well. Work had been done to allow for `timeSeries` objects to be used as well, but many issues were still being worked out. With `xts` now used internally, it is possible to use *any* of R's time-series classes. Simultaneously saving development time and reducing the learning/using curve for the end user. The function now simply handles whatever time-series object it receives exactly as the user expects — without complaint. More details, as well as examples of incorporating `xts` into functions will be covered later in this document.

While it may seem that `xts` is primarily a tool to help make existing R code more user-friendly, the opportunity to add exciting (to software people) new functionality could not be passed up. To this end, `xts` offers the user the ability to add custom attributes to any object — during its construction or at any time thereafter. Additionally, by requiring that the index attribute be derived from one of R's existing time-based classes, `xts` methods can make assumptions, while subsetting by time or date, that allow for much cleaner and accurate data manipulation.

The remainder of this introduction will examine what an `xts` object consists of and its basic usage, explain how developing with `xts` can save package development time, and finally will demonstrate how to extend the class - informally and formally.

2 The structure of `xts`

To understand a bit more of *what an xts object can do*, it may be beneficial to know *what an xts object is*. This section is intended to provide a quick overview of the basics of the class, as well as what features make it unique.

2.1 It's a `zoo` in here

At the core of an `xts` object is a `zoo` object from the package of the same name. Simplified, this class contains an array of values comprising your data (often in matrix form) and an index attribute to provide information about the data's ordering. Most of the details surrounding zoo objects apply equally to xts. As it would be redundent to simply retell the excellent introductory zoo vignette, the reader is advised to read, absorb, and re-read that documentation to best understand the power of this class. The authors of the `xts` package recognize that `zoo`'s strength comes from its simplicity of use, as well as its overall flexibility. What motivated the `xts` extension was a desire to have even more flexibility, while imposing reasonable constraints to make this class into a true time-based one.

2.2 `xts` modifications

Objects of class `xts` differ from objects of class `zoo` in three key ways: the use of formal time-based classes for indexing, internal xts properties, and perhaps most uniquely — user-added attributes.

True time-based indexes

To allow for functions that make use of `xts` objects as a general time-series object - it was necessary to impose a simple rule on the class. The index of each `xts` object *must* be of a known and supported time or date class. At present this includes any one of the following - Date, POSIXct, chron, yearmon, yearqtr, or timeDate. The relative merits of each are left to the judgement of the user, though the first three are expected to be sufficient for most applications.

Internal attributes: .CLASS, .ROWNAMES, etc.

In order for one major feature of the `xts` class to be possible - the conversion and re-conversion of classes to and from `xts` - certain elements must be preserved within the converted object. These are for internal use, and as such require little further explanation in an introductory document. Interested readers are invited to examine the source as well as read the developer documentation.

xtsAttributes

This is what makes the xts class an *extensible* time-series class. Arbitrary attributes may be assigned and removed from the object without causing issues with the data's display or otherwise. Additionally this is where *other* class specific attributes (e.g. *FinCenter* from `timeSeries`) are stored during conversion to an xts object so they may be restored with `reclass`.

3 Using the `xts` package

Just what is required to start using `xts`? Nothing more than a simple conversion of your current time-series data with `as.xts`, or the creation of a new object with the `xts` constructor.

3.1 Creating data objects: `as.xts` and `xts`

There are two equally valid mechanisms to create an `xts` object - coerce a supported time-series class to `xts` with a call to `as.xts` or create a new object from scratch with `xts`.

Converting your *existing* time-series data: `as.xts`

If you are already comfortable using a particular time-series class in `R`, you can still access the functionality of `xts` by converting your current objects.

Presently it is possible to convert all the major time-series like classes in `R` to `xts`. This list includes objects of class: matrix, data.frame, ts, zoo, irts, and timeSeries. The new object will maintain all the necessary information needed to `reclass` this object back to its original class if that is desired. Most classes after re-conversion will be identical to similar modifications on the original object, even after sub-setting or other changes while an `xts` object.

```
> require(xts)
> data(sample_matrix)
> class(sample_matrix)

[1] "matrix" "array"

> str(sample_matrix)

 num [1:180, 1:4] 50 50.2 50.4 50.4 50.2 ...
 - attr(*, "dimnames")=List of 2
  ..$ : chr [1:180] "2007-01-02" "2007-01-03" "2007-01-04" "2007-01-05" ...
  ..$ : chr [1:4] "Open" "High" "Low" "Close"

> matrix_xts <- as.xts(sample_matrix,dateFormat='Date')
> str(matrix_xts)

An ‘xts’ object on 2007-01-02/2007-06-30 containing:
  Data: num [1:180, 1:4] 50 50.2 50.4 50.4 50.2 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Open" "High" "Low" "Close"
  Indexed by objects of class: [Date] TZ: UTC
  xts Attributes:
 NULL
```

```
> df_xts <- as.xts(as.data.frame(sample_matrix),
+  important='very important info!')
> str(df_xts)
```

```
An ‘xts’ object on 2007-01-02/2007-06-30 containing:
  Data: num [1:180, 1:4] 50 50.2 50.4 50.4 50.2 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Open" "High" "Low" "Close"
  Indexed by objects of class: [POSIXct,POSIXt] TZ:
  xts Attributes:
List of 1
 $ important: chr "very important info!"
```

A few comments about the above. `as.xts` takes different arguments, depending on the original object to be converted. Some classes do not contain enough information to infer a time-date class. If that is the case, POSIXct is used by default. This is the case with both matrix and data.frame objects. In the preceding examples we first requested that the new date format be of type 'Date'. The second example was left to the default `xts` method with a custom attribute added.

Creating new data: the `xts` constructor

Data objects can also be constructed directly from raw data with the `xts` constructor function, in essentially the same way a `zoo` object is created with the exception that at present there is no equivelant `zooreg` class.

```
> xts(1:10, Sys.Date()+1:10)
```

```
           [,1]
2020-09-10    1
2020-09-11    2
2020-09-12    3
2020-09-13    4
2020-09-14    5
2020-09-15    6
2020-09-16    7
2020-09-17    8
2020-09-18    9
2020-09-19   10
```

3.2 `xts` methods

There is a full complement of standard methods to make use of the features present in `xts` objects. The generic methods currently extended to `xts` include

“`[`”, `cbind`, `rbind`, `c`, `str`, `Ops`, `print`, `na.omit`, `time`, `index`, `plot` and `coredata`. In addition, most methods that can accept zoo or matrix objects will simply work as expected.

A quick tour of some of the methods leveraged by `xts` will be presented here, including subsetting via “`[`”, indexing objects with `tclass` and `convertIndex`, and a quick look at plotting `xts` objects with the `plot` function.

Subsetting

The most noticable difference in the behavior of `xts` objects will be apparent in the use of the “`[`” operator. Using special notation, one can use date-like strings to extract data based on the time-index. Using increasing levels of time-detail, it is possible to subset the object by year, week, days - or even seconds.

The *i* (row) argument to the subset operator “`[`”, in addition to accepting numeric values for indexing, can also be a character string, a time-based object, or a vector of either. The format must left-specified with respect to the standard ISO:8601 time format — *“CCYY-MM-DD HH:MM:SS”* [4]. This means that for one to extract a particular month, it is necesssary to fully specify the year as well. To identify a particular hour, say all observations in the eighth hour on January 1, 2007, one would likewise need to include the full year, month and day - e.g. “2007-01-01 08”.

It is also possible to explicitly request a range of times via this index-based subsetting, using the ISO-recommended “/” as the range seperater. The basic form is *“from/to”*, where both *from* and *to* are optional. If either side is missing, it is interpretted as a request to retrieve data from the beginning, or through the end of the data object.

Another benefit to this method is that exact starting and ending times need not match the underlying data - the nearest available observation will be returned that is within the requested time period.

The following example shows how to extract the entire month of March 2007 - without having to manually identify the index positions or match the underlying index type. The results have been abbreviated to save space.

```
> matrix_xts['2007-03']
```

```
               Open     High      Low    Close
2007-03-01 50.81620 50.81620 50.56451 50.57075
2007-03-02 50.60980 50.72061 50.50808 50.61559
2007-03-03 50.73241 50.73241 50.40929 50.41033
2007-03-04 50.39273 50.40881 50.24922 50.32636
2007-03-05 50.26501 50.34050 50.26501 50.29567
```

```
...
```

Now extract all the data from the beginning through January 7, 2007.

```
> matrix_xts['/2007-01-07']
```

```
               Open     High      Low    Close
2007-01-02 50.03978 50.11778 49.95041 50.11778
2007-01-03 50.23050 50.42188 50.23050 50.39767
2007-01-04 50.42096 50.42096 50.26414 50.33236
2007-01-05 50.37347 50.37347 50.22103 50.33459
2007-01-06 50.24433 50.24433 50.11121 50.18112
2007-01-07 50.13211 50.21561 49.99185 49.99185
```

Additional xts tools providing subsetting are the `first` and `last` functions. In the spirit of head and tail from the *utils* recommended package, they allow for string based subsetting, without forcing the user to conform to the specifics of the time index, similar in usage to the *by* arguments of `aggregate.zoo` and `seq.POSIXt`.

Here is the first 1 week of the data

```
> first(matrix_xts,'1 week')
```

```
               Open     High      Low    Close
2007-01-02 50.03978 50.11778 49.95041 50.11778
2007-01-03 50.23050 50.42188 50.23050 50.39767
2007-01-04 50.42096 50.42096 50.26414 50.33236
2007-01-05 50.37347 50.37347 50.22103 50.33459
2007-01-06 50.24433 50.24433 50.11121 50.18112
2007-01-07 50.13211 50.21561 49.99185 49.99185
```

...and here is the first 3 days of the last week of the data.

```
> first(last(matrix_xts,'1 week'),'3 days')
```

```
               Open     High      Low    Close
2007-06-25 47.20471 47.42772 47.13405 47.42772
2007-06-26 47.44300 47.61611 47.44300 47.61611
2007-06-27 47.62323 47.71673 47.60015 47.62769
```

Indexing

While the subsetting ability of the above makes exactly *which* time-based class you choose for your index a bit less relevant, it is none-the-less a factor that is beneficial to have control over.

To that end, `xts` provides facilities for indexing based on any of the current time-based classes. These include `Date`, `POSIXct`, `chron`, `yearmon`, `yearqtr`, and `timeDate`. The index itself may be accessed via the zoo generics extended to xts — `index` and the replacement function `index<-`.

It is also possible to directly query and set the index class of an `xts` object by using the respective functions `tclass` and `tclass<-`. Temporary conversion, resulting in a new object with the requested index class, can be accomplished via the `convertIndex` function.

```
> tclass(matrix_xts)

[1] "Date"

> tclass(convertIndex(matrix_xts,'POSIXct'))

[1] "POSIXct" "POSIXt"
```

Plotting

The use of time-based indexes within `xts` allows for assumptions to be made regarding the x-axis of plots. The `plot` method makes use of the `xts` function `axTicksByTime`, which heuristically identifies suitable tickmark locations for printing given a time-based object.

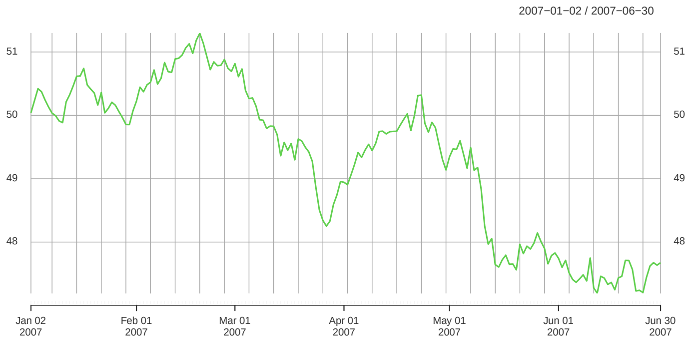
When `axTickByTime` is called with its `ticks.on` argument set to “auto”, the result is a vector of suitably chosen tickmark locations. One can also specify the specific points to use by passing a character string to the argument indicating which time period to create tickmarks on.

```
> axTicksByTime(matrix_xts, ticks.on='months')

Jan 02\n2007 Feb 01\n2007 Mar 01\n2007 Apr 01\n2007 May 01\n2007 Jun 01\n2007
           1           31           59           90          120          151
Jun 30\n2007
         180
```

A simple example of the plotting functionality offered by this labelling can be seen here:

```
> plot(matrix_xts[,1],major.ticks='months',minor.ticks=FALSE,main=NULL,col=3)
```



3.3 Restoring the original class - `reclass` & `Reclass`

By now you may be interested in some of the xts functionality presented, and wondering how to incorporate it into a current workflow — but not yet ready to commit to using it exclusively.

If it is desirable to only use the subsetting tools for instance, a quick conversion to xts via `as.xts` will allow full access to the above subsetting tools. When it is then necessary to continue your analysis using the original class, it is as simple as calling the function `reclass` to return the object to its original class.

(Re)converting classes manually: `reclass`

```
> # using xts-style subsetting doesn't work on non-xts objects
> sample_matrix['2007-06']

[1] NA

> # convert to xts to use time-based subsetting
> str(as.xts(sample_matrix)['2007-06'])

An ‘xts’ object on 2007-06-01/2007-06-30 containing:
  Data: num [1:30, 1:4] 47.7 47.6 47.7 47.5 47.4 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Open" "High" "Low" "Close"
  Indexed by objects of class: [POSIXct,POSIXt] TZ:
  xts Attributes:
 NULL

> # reclass to get to original class back
> str(reclass(as.xts(sample_matrix)['2007-06']))

An ‘xts’ object on 2007-06-01/2007-06-30 containing:
  Data: num [1:30, 1:4] 47.7 47.6 47.7 47.5 47.4 ...
 - attr(*, "dimnames")=List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Open" "High" "Low" "Close"
  Indexed by objects of class: [POSIXct,POSIXt] TZ:
  xts Attributes:
 NULL
```

This differs dramatically from the standard `as.*****` conversion though. Internally, key attributes of your original data object are preserved and adjusted to assure that the process introduces no changes other than those requested. Think of it as a smart `as`.

Behind the scenes, `reclass` has enormous value in functions that convert all incoming data to `xts` for simplified processing. Often it is necessary to return an object back to the user in the class he is expecting — following the principal of least surprise. It is in these circumstances where `reclass` can turn hours of tedious development into mere minutes per function. More details on the details of using this functionality for developers will be covered in section 4, **Developing with xts**.

A user friendly interface of this *reclass* functionality, though implicit, is available in the `Reclass` function. It's purpose is to make it easy to preserve an object's attributes after calling a function that is not programmed to be aware of your particular class.

Letting xts handle the details: `Reclass`

If the function you require does not make use of `reclass` internally, it may still be possible to let xts convert and reconvert your time-based object for you. The caveat here is that the object returned:

- must be of the same length as the first argument to the function.
- intended to be coerced back to the class of the first argument

Simply wrapping the function that meets these criteria in `Reclass` will result in an attempt to coerce the returned output of the function

```
> z <- zoo(1:10,Sys.Date()+1:10)
> # filter converts to a ts object - and loses the zoo class
> (zf <- filter(z, 0.2))

Time Series:
Start = 18515
End = 18524
Frequency = 1
 [1] 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0

> class(zf)

[1] "ts"

> # using Reclass, the zoo class is preserved
> (zf <- Reclass(filter(z, 0.2)))

2020-09-10 2020-09-11 2020-09-12 2020-09-13 2020-09-14 2020-09-15 2020-09-16
       0.2        0.4        0.6        0.8        1.0        1.2        1.4
2020-09-17 2020-09-18 2020-09-19
       1.6        1.8        2.0

> class(zf)

[1] "zoo"
```

The `Reclass` function is still a bit experimental, and will certainly improve in time, but for now provides at least an alternative option to maintain your object's class and attributes when the function you require can't on its own.

3.4 Additional time-based tools

In addition to the core `xts` tools covered above, there are more functions that are included in xts to make the process of dealing with time-series data easier. Some of these have been moved from the package `quantmod` to `xts` to make it easier to use them within other applications.

Calculate periodicity

The `periodicity` function provides a quick summary as to the underlying periodicity of most time-series like objects. Primarily a wrapper to `difftime` it provides a quick and concise summary of your data.

```
> periodicity(matrix_xts)
```

```
Daily periodicity from 2007-01-02 to 2007-06-30
```

Find endpoints by time

Another common issue with time-series data is identifying the endpoints with respect to time. Often it is necessary to break data into hourly or monthly intervals to calculate some statistic. A simple call to `endpoints` offers a quick vector of values suitable for subsetting a dataset by. Note that the first element it zero, which is used to delineate the *end*.

```
> endpoints(matrix_xts,on='months')
```

```
[1]   0  30  58  89 119 150 180
```

```
> endpoints(matrix_xts,on='weeks')
```

```
 [1]   0   6  13  20  27  34  41  48  55  62  69  76  83  90  97 104 111 118 125
[20] 132 139 146 153 160 167 174 180
```

Change periodicity

One of the most ubiquitous type of data in finance is OHLC data (Open-High-Low-Close). Often is is necessary to change the periodicity of this data to something coarser - e.g. take daily data and aggregate to weekly or monthly. With `to.period` and related wrapper functions it is a simple proposition.

```
> to.period(matrix_xts,'months')
```

```
           matrix_xts.Open matrix_xts.High matrix_xts.Low matrix_xts.Close
2007-01-31        50.03978        50.77336       49.76308         50.22578
2007-02-28        50.22448        51.32342       50.19101         50.77091
2007-03-31        50.81620        50.81620       48.23648         48.97490
2007-04-30        48.94407        50.33781       48.80962         49.33974
2007-05-31        49.34572        49.69097       47.51796         47.73780
2007-06-30        47.74432        47.94127       47.09144         47.76719
```

```
> periodicity(to.period(matrix_xts,'months'))
```

```
Monthly periodicity from 2007-01-31 to 2007-06-30
```

```
> # changing the index to something more appropriate
> to.monthly(matrix_xts)
```

```
         matrix_xts.Open matrix_xts.High matrix_xts.Low matrix_xts.Close
Jan 2007        50.03978        50.77336       49.76308         50.22578
Feb 2007        50.22448        51.32342       50.19101         50.77091
Mar 2007        50.81620        50.81620       48.23648         48.97490
Apr 2007        48.94407        50.33781       48.80962         49.33974
May 2007        49.34572        49.69097       47.51796         47.73780
Jun 2007        47.74432        47.94127       47.09144         47.76719
```

The `to.monthly` wrapper automatically requests that the returned object have an index/rownames using the `yearmon` class. With the `indexAt` argument it is possible to align most series returned to the end of the period, the beginning of the period, or the first or last observation of the period — even converting to something like `yearmon` is supported. The online documentation provides more details as to additional arguments.

Periodically apply a function

Often it is desirable to be able to calculate a particular statistic, or evaluate a function, over a set of non-overlapping time periods. With the `period.apply` family of functions it is quite simple.

The following examples illustrate a simple application of the `max` function to our example data.

```
> # the general function, internally calls sapply
> period.apply(matrix_xts[,4],INDEX=endpoints(matrix_xts),FUN=max)

              Close
2007-01-31 50.67835
2007-02-28 51.17899
2007-03-31 50.61559
2007-04-30 50.32556
2007-05-31 49.58677
2007-06-30 47.76719

> # same result as above, just a monthly interface
> apply.monthly(matrix_xts[,4],FUN=max)

              Close
2007-01-31 50.67835
2007-02-28 51.17899
2007-03-31 50.61559
2007-04-30 50.32556
2007-05-31 49.58677
2007-06-30 47.76719

> # using one of the optimized functions - about 4x faster
> period.max(matrix_xts[,4], endpoints(matrix_xts))
```

```
                [,1]
2007-01-31 50.67835
2007-02-28 51.17899
2007-03-31 50.61559
2007-04-30 50.32556
2007-05-31 49.58677
2007-06-30 47.76719
```

In addition to `apply.monthly`, there are wrappers to other common time frames including: `apply.daily`, `apply.weekly`, `apply.quarterly`, and `apply.yearly`. Current optimized functions include `period.max`, `period.min`, `period.sum`, and `period.prod`.

4 Developing with `xts`

While the tools available to the xts *user* are quite useful, possibly greater utility comes from using xts internally as a *developer*. Bypassing traditional S3/S4 method dispatch and custom if-else constructs to handle different time-based classes, `xts` not only makes it easy to handle all supported classes in one consistent manner, it also allows the whole process to be invisible to the function user.

4.1 One function for all classes: `try.xts`

With the proliferation of data classes in R, it can be tedious, if not entirely impractical, to manage interfaces to all classes.

Not only does trying to handle every possible class present non-trivial design issues, the developer is also forced to learn and understand the nuances of up to eight or more classes. For each of these classes it is then ncessary to write and manage corresponding methods for each case.

At best, this reduces the time available to devote to core function functionality — at worst is a prime opportunity to introduce errors that inevitibly come from this massive increase in code.

The solution to this issue is to use one class internally within your package, or more generally your entire workflow. This can be accomplished in one of two ways: force your users to adopt the convention you do, or allow for multiple object classes by relying on internal code to convert to one consistent type.

Using the second approach offers the most end-user flexibility, as class conversions are no longer required simply to make use of package functionality. The user's own workflow need not be interrupted with unproductive and potentially error-prone conversions and reconversions.

Using the functionality of `try.xts` and `reclass` offered by the xts package allows the developer an opportunity to cleanly, and reliably, manage data with the least amount of code, and the least number of artificial end-user restrictions.

An example from the xts package illustrates just how simple this can be.

```
> period.apply

function (x, INDEX, FUN, ...)
{
    x <- try.xts(x, error = FALSE)
    FUN <- match.fun(FUN)
    if (!isOrdered(INDEX)) {
        INDEX <- sort(unique(INDEX))
    }
    if (INDEX[1] != 0) {
        INDEX <- c(0, INDEX)
    }
    if (last(INDEX) != NROW(x)) {
```

```
        INDEX <- c(INDEX, NROW(x))
    }
    xx <- sapply(1:(length(INDEX) - 1), function(y) {
        FUN(x[(INDEX[y] + 1):INDEX[y + 1]], ...)
    })
    if (is.vector(xx))
        xx <- t(xx)
    xx <- t(xx)
    if (is.null(colnames(xx)) && NCOL(x) == NCOL(xx))
        colnames(xx) <- colnames(x)
    reclass(xx, x[INDEX])
}
<bytecode: 0x55e4188b1c28>
<environment: namespace:xts>
```

Some explanation of the above code is needed. The `try.xts` function takes three arguments, the first is the object that the developer is trying to convert, the second . . . is any additional arguments to the `as.xts` constructor that is called internally (ignore this for the most part — though it should be noted that this is an R dots argument . . .), and the third is a what the result of an error should be.

Of the three, `error` is probably the most useful from a design standpoint. Some functions may not be able to deal with data that isn't time-based. Simple numerical vectors might not contain enough information to be of any use. The *error* argument lets the developer decide if the function should be halted at this point, or continue onward. If a logical value, the result is handled by R's standard error mechanism during the try-catch block of code internal to `try.xts`. If error is a character string, it is returned to the standard output as the message. This allows for diagnostic messages to be fine tuned to your particular application.

The result of this call, if successful (or if `error=FALSE`) is an object that may be of class `xts`. If your function can handle either numeric data or time-based input, you can branch code here for cases you expect. If your code has been written to be more general at this point, you can simply continue with your calculations, the originally converted object will contain the information that will be required to reclass it at the end.

A note of importance here: if you are planning on returning an object that is of the original class, it is important to not modify the originally coverted object - in this case that would be the `x` result of the `try.xts` call. You will notice that the function's result is assigned to `xx` so as not to impact the original converted function. If this is not possible, it is recommended to copy the object first to preserve an untouched copy for use in the `reclass` function.

Which leads to the second part of the process of developing with xts.

4.2 Returning the original class: `reclass`

The `reclass` function takes the object you are expecting to return to your user (the result of all your calculations) and optionally an `xts` object that was the result of the original `try.xts` call.

It is important to stress that the `match.to` object *must be an untouched object* returned from the `try.xts` call. The only exception here is when the resultant data has changed dimension — as is the case in the `period.apply` example. As reclass will try and convert the first argument to the orginal class of the second (the original class passed to the function), it must have the same general row dimension of the original.

A final note on using `reclass`. If the `match.to` argument is left off, the conversion will only be attempted if the object is of class `xts` and has a `CLASS` attribute that is not `NULL`, else the object is simply returned. Essentially if the object meant to be reconverted is already of in the form needed by the individual reclass methods, generally nothing more needs to be done by the developer.

In many cases your function does not need to return an object that is expected to be used in the same context as the original. This would be the case for functions that summarize an object, or perform some statistical analysis.

For functions that do not need the `reclass` functionality, a simple use of `try.xts` at the beginning of the function is all that is needed to make use of this single-interface tool within `xts`.

Further examples can be found in the `xts` functions `periodicity` and `endpoints` (no use of reclass), and `to.period` (returns an object of the original's class). The package `quantmod` also utilizes the `try.xts` functionality in its `chartSeries` function — allowing financial charts for all time-based classes.

Forthcoming developer documentation will examine the functions highlighted above, as well go into more detail on exceptional cases and requirements.

5 Customizing and Extending xts

As *extensible* is in the name of the package, it is only logical that it can be extended. The two obvious mechanisms to make `xts` match the individual needs of a diverse user base is the introduction of custom attributes, and the idea of subclassing the entire `xts` class.

5.1 `xtsAttributes`

What makes an R attribute an `xtsAttribute`? Beyond the sematics, xtsAttributes are designed to persist once attached to an object, as well as not get in the way of other object functionality. All xtsAttributes are indeed R attributes, though the same can not be said of the reverse — all R attributes are *not* xtsAttributes!

Attaching arbitrary attributes to most (all?) classes other than `xts` will cause the attribute to be displayed during most calls that print the object. While this isn't necessarily devestating, it is often time unsightly, and sometimes even confusing to the end user (this may depend on the quality your users).

xts offers the developer and end-user the opportunity to attach attributes with a few different mechanisms - and all will be suppressed from normal view, unless specifically called upon.

What makes an xtsAttribute special is that it is principally a mechanism to store and view meta-data, that is, attributes that would be seen with a call to R's `attributes`.

```
> str(attributes(matrix_xts))

List of 4
 $ dim     : int [1:2] 180 4
 $ dimnames:List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Open" "High" "Low" "Close"
 $ index   : num [1:180] 1.17e+09 1.17e+09 1.17e+09 1.17e+09 1.17e+09 ...
  ..- attr(*, "tzone")= chr "UTC"
  ..- attr(*, "tclass")= chr "Date"
 $ class   : chr [1:2] "xts" "zoo"

> str(xtsAttributes(matrix_xts))

 NULL

> # attach some attributes
> xtsAttributes(matrix_xts) <- list(myattr="my meta comment")
> attr(matrix_xts, 'another.item') <- "one more thing..."
> str(attributes(matrix_xts))

List of 6
 $ dim         : int [1:2] 180 4
```

```
 $ dimnames    :List of 2
  ..$ : NULL
  ..$ : chr [1:4] "Open" "High" "Low" "Close"
 $ index       : num [1:180] 1.17e+09 1.17e+09 1.17e+09 1.17e+09 1.17e+09 ...
  ..- attr(*, "tzone")= chr "UTC"
  ..- attr(*, "tclass")= chr "Date"
 $ class       : chr [1:2] "xts" "zoo"
 $ myattr      : chr "my meta comment"
 $ another.item: chr "one more thing..."
```

```
> str(xtsAttributes(matrix_xts))
```

```
List of 2
 $ myattr      : chr "my meta comment"
 $ another.item: chr "one more thing..."
```

In general - the only attributes that should be handled directly by the user (*without* the assistance of xts functions) are ones returned by `xtsAttributes`. The additional attributes seen in the `attributes` example are for internal R and xts use, and if you expect unbroken code, should be left alone.

5.2 Subclassing `xts`

Subclassing xts is as simple as extending any other S3 class in R. Simply include the full class of the xts system in your new class.

```
> xtssubclass <- structure(matrix_xts, class=c('xts2','xts','zoo'))
> class(xtssubclass)
```

```
[1] "xts2" "xts"  "zoo"
```

This will allow the user to override methods of xts and zoo, while still allowing for backward compatibility with all the tools of xts and zoo, much the way `xts` benefits from extending `zoo`.

6 Conclusion

The `xts` package offers both R developers and R users an extensive set of time-aware tools for use in time-based applications. By extending the `zoo` class, xts leverages an excellent infrastructure tool into a true time-based class. This simple requirement for time-based indexing allows for code to make assumptions about the object's purpose, and facilitates a great number of useful utilities — such as time-based subsetting.

Additionally, by embedding knowledge of the currently used time-based classes available in R, xts can offer the developer and end-user a single interface mechanism to make internal class decisions user-driven. This affords developers an opportunity to design applications for there intended purposes, while freeing up time previously used to manage the data structures.

Future development of xts will focus on integrating xts into more external packages, as well as additional useful additions to the time-based utilities currently available within the package. An effort to provide external disk and memory based data access will also be examined for potential inclusion or extension.

References

[1] Achim Zeileis and Gabor Grothendieck (2005): *zoo: S3 Infrastructure for Regular and Irregular Time Series.* Journal of Statistical Software, 14(6), 1-27. URL http://www.jstatsoft.org/v14/i06/

[2] Adrian Trapletti and Kurt Hornik (2007): *tseries: Time Series Analysis and Computational Finance.* R package version 0.10-11.

[3] Diethelm Wuertz, many others and see the SOURCE file (2007): *Rmetrics: Rmetrics - Financial Engineering and Computational Finance.* R package version 260.72. http://www.rmetrics.org

[4] International Organization for Standardization (2004): *ISO 8601: Data elements and interchage formats — Information interchange — Representation of dates and time* URL http://www.iso.org

[5] R Development Core Team: *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org

[6] Jeffrey A. Ryan (2008): *quantmod: Quantitative Financial Modelling Framework.* R package version 0.3-5. URL http://www.quantmod.com URL http://r-forge.r-project.org/projects/quantmod