FRE6871 R in Finance

Lecture#6, Fall 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 15, 2024

Flattening a List of Vectors to a Matrix Using do.call()

A list of vectors can be flattened into a matrix using the functions do.call() and either rbind() or cbind().

If the list contains vectors of different lengths, then R applies the recycling rule.

If the list contains a NULL element, that element is skipped.

```
> # Create list of vectors
> listv <- lapply(1:3, function(x) sample(6))
> # Bind list elements into matrix - doesn't work
> rbind(listv)
> # Bind list elements into matrix - tedious
> rbind(listv[[1]], listv[[2]], listv[[3]])
> # Bind list elements into matrix - works!
> do.call(rbind, listv)
> # Create numeric list
> listv <- list(1, 2, 3, 4)
> do.call(rbind, listv)  # Returns single column matrix
> do.call(cbind, listv)  # Returns single row matrix
> # Recycling rule applied
> do.call(cbind, list(1:2, 3:5))
> # NULL element is skipped
> do.call(cbind, list(1, NULL, 3, 4))
> # NA element isn't skipped
> do.call(cbind, list(1, NA, 3, 4))
```

Efficient Binding of Lists Into Matrices

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

But for large vectors this procedure can be very slow, and often causes an out of memory error.

The function `do_call_rbind()` efficiently combines a list of vectors into a matrix.

`do_call_rbind()` produces the same result as `do.call(rbind, list_var)`, but using recursion.

`do_call_rbind()` calls lapply in a loop, each time binding neighboring list elements and dividing the length of the list by half.

`do_call_rbind()` is the same function as `do.call.rbind()` from package *qmao*:
https://r-forge.r-project.org/R/?group_id=1113

```
> listv <- lapply(1:5, rnorm, n=10)
> matv <- do.call(rbind, listv)
> dim(matv)
> do_call_rbind <- function(listv) {
+   while (NROW(listv) > 1) {
+ # Index of odd list elements
+     odd_index <- seq(from=1, to=NROW(listv), by=2)
+ # Bind odd and even elements, and divide listv by half
+     listv <- lapply(odd_index, function(indeks) {
+ if (indeks==NROW(listv)) return(listv[[indeks]])
+ rbind(listv[[indeks]], listv[[indeks+1]])
+     })  # end lapply
+   }  # end while
+ # listv has only one element - return it
+   listv[[1]]
+ }  # end do_call_rbind
> all.equal(matv, do_call_rbind(listv))
```

Filtering Data Frames Using `subset()`

Filtering means extracting rows from a *data frame* that satisfy a logical condition.

Data frames can be filtered using Boolean vectors and brackets `"[]"` operators.

The function `subset()` filters *data frames*, by applying logical conditions to its columns, using the column names.

`subset()` provides a succinct notation and discards `NA` values, but it's slightly slower than using `Boolean` vectors and brackets `"[]"` operators.

```
> airquality[(airquality$Solar.R > 320 & !is.na(airquality$Solar.R)
> subset(x=airquality, subset=(Solar.R > 320))
> summary(microbenchmark(
+     subset=subset(x=airquality, subset=(Solar.R > 320)),
+     brackets=airquality[(airquality$Solar.R > 320 &
+             !is.na(airquality$Solar.R)), ],
+ times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Splitting Data Frames Using `factor` Categorical Variables

The function `split()` divides an object into a list of objects, according to a `factor` (categorical variable).

The list's `names` attribute is equal to the `factor` levels.

```
> unique(iris$Species)  # Species has three distinct values
> # Split into separate data frames by hand
> setosa <- iris[iris$Species=="setosa", ]
> versi <- iris[iris$Species=="versicolor", ]
> virgin <- iris[iris$Species=="virginica", ]
> dim(setosa)
> head(setosa, 2)
> # Split iris into list based on Species
> splitiris <- split(iris, iris$Species)
> str(splitiris, max.confl=1)
> names(splitiris)
> dim(splitiris$setosa)
> head(splitiris$setosa, 2)
> all.equal(setosa, splitiris$setosa)
```

The *split-apply-combine* Procedure

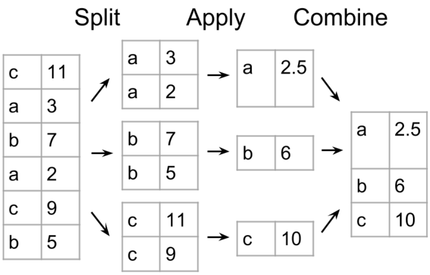
The *split-apply-combine* procedure consists of:

- dividing an object into a list, according to a factor (attribute).
- applying a function to each list element.
- combining the results.

The *split-apply-combine* procedure is also called the *map-reduce* procedure, or simply *data pivoting*, and it's similar to *pivot tables* in *Excel*.

Data pivoting can be performed *data frames*, by aggregating its columns based on categorical data stored in one of its columns.

You can read more about the *split-apply-combine* procedure in Hadley Wickham's paper:
http://www.jstatsoft.org/v40/i01/paper



Data Pivoting Example

Data pivoting can be performed through successive applications of functions `split()`, `apply()`, and `unlist()`.

A *data frame* can be *pivoted* either by first splitting it into a list of *data frames* and then aggregating, or by splitting just a single column and aggregating it.

The function `split()` divides an object into a list of objects, according to a `factor` (categorical variable).

The list's `names` attribute is equal to the `factor` levels.

The functional `aggregate()` *pivots* the columns of a *data frame*.

`aggregate()` can accept a `"formula"` argument with the column names, or it can accept `"x"` and `"by"` arguments with the columns.

`aggregate()` returns a *data frame* containing the names of the groups (`factor` confls).

```
> unique(mtcars$cyl)  # cyl has three unique values
> # Split mpg column based on number of cylinders
> split(mtcars$mpg, mtcars$cyl)
> # Split mtcars data frame based on number of cylinders
> split_cars <- split(mtcars, mtcars$cyl)
> str(split_cars, max.confl=1)
> names(split_cars)
> # Aggregate the mean mpg over split mtcars data frame
> sapply(split_cars, function(x) mean(x$mpg))
> # Or: split mpg column and aggregate the mean
> sapply(split(mtcars$mpg, mtcars$cyl), mean)
> # Same but using with()
> with(mtcars, sapply(split(mpg, cyl), mean))
> # Or: aggregate() using formula syntax
> aggregate(x=(mpg ~ cyl), data=mtcars, FUN=mean)
> # Or: aggregate() using data frame syntax
> aggregate(x=mtcars$mpg, by=list(cyl=mtcars$cyl), FUN=mean)
> # Or: using name for mpg
> aggregate(x=list(mpg=mtcars$mpg), by=list(cyl=mtcars$cyl), FUN=mea
> # Aggregate() all columns
> aggregate(x=mtcars, by=list(cyl=mtcars$cyl), FUN=mean)
> # Aggregate multiple columns using formula syntax
> aggregate(x=(cbind(mpg, hp) ~ cyl), data=mtcars, FUN=mean)
```

The tapply() Functional

The functional `tapply()` is a specialized version of the `apply()` functional, that applies a function to elements of a *jagged array*.

A *jagged array* is a list consisting of vectors or matrices of different lengths.

`tapply()` accepts a vector of values `"X"`, a factor `"INDEX"`, and a function `"FUN"`.

`tapply()` first groups the elements of `"X"` according to the factor `"INDEX"`, transforming it into a *jagged array*, and then applies `"FUN"` to each element of the *jagged array*.

`tapply()` applies a function to sub-vectors aggregated using a factor, and performs *data pivoting* in a single function call.

The `by()` function is a wrapper for `tapply()`.

The `with()` function evaluates an expression in an environment constructed from the data.

```
> # Mean mpg for each cylinder group
> tapply(X=mtcars$mpg, INDEX=mtcars$cyl, FUN=mean)
> # using with() environment
> with(mtcars, tapply(X=mpg, INDEX=cyl, FUN=mean))
> # Function sapply() instead of tapply()
> with(mtcars, sapply(sort(unique(cyl)), function(x) {
+  structure(mean(mpg[x==cyl]), names=x)
+      }))  # end with
> # Function by() instead of tapply()
> with(mtcars, by(data=mpg, INDICES=cyl, FUN=mean))
```

Data Pivoting Returning a Matrix

Sometimes *data pivoting* returns a list of vectors.

A list of vectors can be flattened into a matrix using the functions `do.call()` and either `rbind()` or `cbind()`.

The function `do.call()` executes a function call using a function name and a list of arguments.

`do.call()` passes the list elements individually, instead of passing the whole list as one argument:
`do.call(fun, list)= fun(list[[1]], list[[2]], ...)`

```
> # Get several mpg stats for each cylinder group
> cardata <- sapply(split_cars, function(x) {
+   c(mean=mean(x$mpg), max=max(x$mpg), min=min(x$mpg))
+ }  # end anonymous function
+ )  # end sapply
> cardata  # sapply() produces a matrix
> # Now same using lapply
> cardata <- lapply(split_cars, function(x) {
+   c(mean=mean(x$mpg), max=max(x$mpg), min=min(x$mpg))
+ }  # end anonymous function
+ )  # end sapply
> is.list(cardata)  # lapply produces a list
> # do.call flattens list into a matrix
> do.call(cbind, cardata)
```

Data Pivoting of Panel Data

The *data frame* panel_data contains fundamental financial data for *S&P500* stocks.

The Industry column has 22 unique elements, while the Sector column has 10 unique elements.

Each Industry belongs to a single Sector, but each Sector may have several Industries that belong to it.

The functional aggregate() allows aggregating over the Industry column, by performing *data pivoting*.

```
> # Download CRSPpanel.txt from the NYU share drive
> # Read the file using read.table() with header and sep arguments
> paneld <- read.table(file="/Users/jerzy/Develop/lecture_slides/dat
+                      header=TRUE, sep="\t")
> paneld[1:5, 1:5]
> attach(paneld)
> # Split paneld based on Industry column
> panelds <- split(paneld, Industry)
> # Number of companies in each Industry
> sapply(panelds, NROW)
> # Number of Sectors that each Industry belongs to
> sapply(panelds, function(x) {
+   NROW(unique(x$Sector))
+ })  # end sapply
> # Or
> aggregate(x=(Sector ~ Industry),
+   data=paneld, FUN=function(x) NROW(unique(x)))
> # Industries and the Sector to which they belong
> aggregate(x=(Sector ~ Industry), data=paneld, FUN=unique)
> # Or
> aggregate(x=Sector, by=list(Industry), FUN=unique)
> # Or
> sapply(unique(Industry), function(x) {
+   Sector[match(x, Industry)]
+ })  # end sapply
```

Data Pivoting Returning a *Jagged Array*

A *jagged array* is a list consisting of vectors or matrices of different lengths.

The functional `aggregate()` returns a *data frame*, so it's output must be coerced if the *data pivoting* attempts to return a *jagged array*.

The functional `tapply()` returns an array, so it's output must be coerced if the *data pivoting* attempts to return a *jagged array*.

`tapply()` accepts a vector of values `"X"`, a factor `"INDEX"`, and a function `"FUN"`.

`tapply()` first groups the elements of `"X"` according to the factor `"INDEX"`, transforming it into a *jagged array*, and then applies `"FUN"` to each element of the *jagged array*.

`tapply()` applies a function to sub-vectors aggregated using a factor, and performs *data pivoting* in a single function call.

```
> # Split paneld based on Sector column
> panelds <- split(paneld, Sector)
> # Number of companies in each Sector
> sapply(panelds, NROW)
> # Industries belonging to each Sector (jagged array)
> secind <- sapply(panelds, function(x) unique(x$Industry))
> # Or use aggregate() (returns a data frame)
> secind2 <- aggregate(x=(Industry ~ Sector),
+   data=paneld, FUN=function(x) unique(x))
> # Or use aggregate() with "by" argument
> secind2 <- aggregate(x=Industry, by=list(Sector),
+     FUN=function(x) as.vector(unique(x)))
> # Coerce secind2 into a jagged array
> namev <- secind2[, 1]
> secind2 <- secind2[, 2]
> names(secind2) <- namev
> all.equal(secind2, secind)
> # Or use tapply() (returns an array)
> secind2 <- tapply(X=Industry, INDEX=Sector, FUN=unique)
> # Coerce secind2 into a jagged array
> secind2 <- drop(as.matrix(secind2))
> all.equal(secind2, secind)
```

Data Pivoting Over Multiple Columns

Data pivoting over multiple columns can be performed by splitting the *data frame* and then performing an sapply() loop using an anonymous function.

Splitting the *data frame* allows aggregations over multiple columns.

An anonymous function allows applying different aggregations on the same column.

```
> # Average ROE in each Industry
> sapply(split(ROE, Industry), mean)
> # Average, min, and max ROE in each Industry
> t(sapply(split(ROE, Industry), FUN=function(x)
+   c(mean=mean(x), max=max(x), min=min(x))))
> # Split paneld based on Industry column
> panelds <- split(paneld, Industry)
> # Average ROE and EPS in each Industry
> t(sapply(panelds, FUN=function(x)
+   c(mean_roe=mean(x$ROE),
+     mean_eps=mean(x$EPS.EXCLUDE.EI))))
> # Or: split paneld based on Industry column
> panelds <- split(paneld[, c("ROE", "EPS.EXCLUDE.EI")],
+   paneld$Industry)
> # Average ROE and EPS in each Industry
> t(sapply(panelds, FUN=function(x) sapply(x, mean)))
> # Average ROE and EPS using aggregate()
> aggregate(x=paneld[, c("ROE", "EPS.EXCLUDE.EI")],
+   by=list(paneld$Industry), FUN=mean)
```

Exception Conditions: Errors and Warnings

Exception conditions are R objects containing information about *errors* or *warnings* produced while evaluating expressions.

The function `warning()` produces a *warning* condition, but doesn't halt function execution, and returns its message to the warning handler.

The function `stop()` produces an *error* condition, halts function execution, and returns its message to the error handler.

The handling of *warning* conditions depends on the value of `options("warn")`:

- *negative* then warnings are ignored,
- *zero* then warnings are stored and printed after the top-confl function has completed,
- *one* - warnings are printed as they occur,
- *two* or larger - warnings are turned into errors,

The function `suppressWarnings()` evaluates its expressions and ignores all warnings.

```
> # ?options  # Get info on global options
> getOption("warn")  # Global option for "warn"
> options("warn")  # Global option for "warn"
> getOption("error")  # Global option for "error"
> calc_sqrt <- function(inputv) {
+ # Returns its argument
+   if (inputv < 0) {
+     warning("calc_sqrt: input is negative")
+     NULL  # Return NULL for negative argument
+   } else {
+     sqrt(inputv)
+   }  # end if
+ }  # end calc_sqrt
> calc_sqrt(5)
> calc_sqrt(-1)
> options(warn=-1)
> calc_sqrt(-1)
> options(warn=0)
> calc_sqrt()
> options(warn=1)
> calc_sqrt()
> options(warn=3)
> calc_sqrt()
```

Validating Function Arguments

Argument validation consists of first determining if any arguments are *missing*, and then determining if the arguments are of the correct *type*.

An argument is *missing* when the formal argument is not bound to an actual value in the function call.

The function `missing()` returns `TRUE` if an argument is missing, and `FALSE` otherwise.

Missing arguments can be detected by:
- assigning a `NULL` default value to formal arguments and then calling `is.null()` on them,
- calling the function `missing()` on the arguments.

The argument *type* can be validated using functions such as `is.numeric()`, `is.character()`, etc.

The function `return()` returns its argument and terminates futher function execution.

```
> # Function valido validates its arguments
> valido <- function(inputv=NULL) {
+ # Check if argument is valid and return double
+   if (is.null(inputv)) {
+     return("valido: input is missing")
+   } else if (is.numeric(inputv)) {
+     2*inputv
+   } else cat("valido: input not numeric")
+ }  # end valido
> valido(3)
> valido("a")
> valido()
> # valido validates arguments using missing()
> valido <- function(inputv) {
+ # Check if argument is valid and return double
+   if (missing(inputv)) {
+     return("valido: input is missing")
+   } else if (is.numeric(inputv)) {
+     2*inputv
+   } else cat("valido: input is not numeric")
+ }  # end valido
> valido(3)
> valido("a")
> valido()
```

Validating Assertions Inside Functions

If assertions about variables inside a function are FALSE, then stop() can be called to halt its execution.

Calling stop() is preferable to calling return(), or inserting cat() statements into the code.

Using stop() inside a function allows calling the function traceback(), if an error was produced.

The function traceback() prints the call stack, showing the function that produced the *error* condition.

cat() statements inside the function body provide information about the state of its variables.

```
> # valido() validates its arguments and assertions
> valido <- function(inputv) {
+ # Check if argument is valid and return double
+   if (missing(inputv)) {
+     stop("valido: input is missing")
+   } else if (!is.numeric(inputv)) {
+     cat("input =", inputv, "\n")
+     stop("valido: input is not numeric")
+   } else 2*inputv
+ }  # end valido
> valido(3)
> valido("a")
> valido()
> # Print the call stack
> traceback()
```

Validating Assertions Using stopifnot()

R provides robust validation and debugging tools through *type* validation functions, and functions missing(), stop(), and stopifnot().

If the argument to function stopifnot() is FALSE, then it produces an *error* condition, and halts function execution.

stopifnot() is a convenience wrapper for stop(), and eliminates the need to use if () statements.

stopifnot() is often used to check the validity of function arguments.

stopifnot() can be inserted anywhere in the function body in order to check assertions about its variables.

```
> valido <- function(inputv) {
+ # Check argument using long form '&&' operator
+   stopifnot(!missing(inputv) && is.numeric(inputv))
+   2*inputv
+ }  # end valido
> valido(3)
> valido()
> valido("a")
> valido <- function(inputv) {
+ # Check argument using logical '&' operator
+   stopifnot(!missing(inputv) & is.numeric(inputv))
+   2*inputv
+ }  # end valido
> valido()
> valido("a")
```

Validating Function Arguments and Assertions

The functions `stop()` and `stopifnot()` halt function execution and produce *error* conditions if certain assertions are `FALSE`.

The *type* validation functions, such as `is.numeric()`, `is.na()`, etc., and `missing()`, allow for validation of arguments and variables inside functions.

`cat()` statements can provide information about the state of variables inside a function.

`cat()` statements don't return values, so they provide information even when a function produces an `error`.

```
> # sumtwo() returns the sum of its two arguments
> sumtwo <- function(input1, input2) {  # Even more robust
+ # Check if at least one argument is not missing
+   stopifnot(!missing(input1) && !missing(input2))
+ # Check if arguments are valid and return sum
+   if (is.numeric(input1) && is.numeric(input2)) {
+     input1 + input2  # Both valid
+   } else if (is.numeric(input1)) {
+     cat("input2 is not numeric\n")
+     input1  # input1 is valid
+   } else if (is.numeric(input2)) {
+     cat("input1 is not numeric\n")
+     input2  # input2 is valid
+   } else {
+     stop("none of the arguments are numeric")
+   }
+ }  # end sumtwo
> sumtwo(1, 2)
> sumtwo(5, 'a')
> sumtwo('a', 5)
> sumtwo('a', 'b')
> sumtwo()
```

The R Debugger Facility

The function debug() flags a function for future debugging, but doesn't invoke the debugger.

After a function is flagged for debugging with the call "debug(myfun)", then the function call "myfun()" automatically invokes the debugger (browser).

When the debugger is first invoked, it prints the function code to the console, and produces a *browser* prompt: "Browse[2]>".

Once inside the debugger, the user can execute the function code one command at a time by pressing the *Enter* key.

The user can examine the function arguments and variables with standard R commands, and can also change the values of objects or create new ones.

The command "c" executes the remainder of the function code without pausing.

The command "Q" exits the debugger (browser).

The call "undebug(myfun)" at the R prompt unflags the function for debugging.

```
> # Flag "valido" for debugging
> debug(valido)
> # Calling "valido" starts debugger
> valido(3)
> # unflag "valido" for debugging
> undebug(valido)
```

Debugging Using browser()

As an alternative to flagging a function for debugging, the user can insert the function `browser()` into the function body.

`browser()` pauses the execution of a function and invokes the debugger (browser) at the point where `browser()` was called.

Once inside the debugger, the user can execute all the same browser commands as when using `debug()`.

`browser()` is usually inserted just before the command that is suspected of producing an *error* condition.

Another alternative to flagging a function for debugging, or inserting `browser()` calls, is setting the `"error"` option equal to `"recover"`.

Setting the `"error"` option equal to `"recover"` automatically invokes the debugger when an *error* condition is produced.

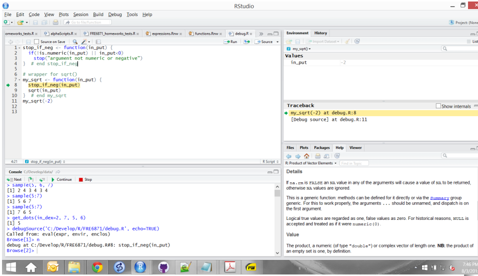
```
> valido <- function(inputv) {
+   browser()  # Pause and invoke debugger
+ # Check argument using long form '&&' operator
+   stopifnot(!missing(inputv) && is.numeric(inputv))
+   2*inputv
+ }  # end valido
> valido()  # Invokes debugger
> options("error")  # Show default NULL "error" option
> options(error=recover)  # Set "error" option to "recover"
> options(error=NULL)  # Set back to default "error" option
```

Using the Debugger in *RStudio*

RStudio has several built-in debugging facilities that complement those already installed in `R`:

- toggling breakpoints, instead of inserting `browser()` commands,
- stepping into functions,
- environment pane with environment stack, instead of calling `ls()`,
- traceback pane, instead of calling `traceback()`,

RStudio provides an online debugging tutorial:
https://support.rstudio.com/hc/en-us/articles/205612627-Debugging-with-RStudio



Handling Exception Conditions

The function `tryCatch()` executes functions and expressions, and handles any *exception conditions* produced when they are evaluated.

`tryCatch()` first evaluates its `"expression"` argument.

If no error or warning `condition` is produced then `tryCatch()` just returns the value of the expression.

If an `exception condition` is produced then `tryCatch()` invokes error and warning *handlers* and executes other expressions to provide information about the `exception condition`.

If a *handler* is provided to `tryCatch()` then the error is captured by the *handler*, instead of being broadcast to the console.

At the end, `tryCatch()` evaluates the expression provided to the `finally` argument.

```
> str(tryCatch)  # Get arguments of tryCatch()
> tryCatch(  # Without error handler
+   {  # Evaluate expressions
+     numv <- 101  # Assign
+     stop("my error")  # Produce error
+   },
+   finally=print(paste0("numv = ", numv))
+ )  # end tryCatch
>
> tryCatch(  # With error handler
+   {  # Evaluate expressions
+     numv <- 101  # Assign
+     stop("my error")  # Produce error
+   },
+   # Error handler captures error condition
+   error=function(msg) {
+     print(paste0("Error handler: ", msg))
+   },  # end error handler
+   # Warning handler captures warning condition
+   warning=function(msg) {
+     print(paste0("Warning handler: ", msg))
+   },  # end warning handler
+   finally=print(paste0("numv = ", numv))
+ )  # end tryCatch
```

Error Conditions in Loops

If an *error* occurs in an `apply()` loop, then the loop exits without returning any result.

`apply()` collects the values returned by the function supplied to its `FUN` argument, and returns them only after the loop is finished.

If one of the function calls produces an error, then the loop is interrupted and `apply()` exits without returning any result.

The function `tryCatch()` captures errors, allowing loops to continue after the error `condition`.

```
> # Apply loop without tryCatch
> apply(matrix(1:5), 1, function(numv) {  # Anonymous function
+     stopifnot(!(numv == 3))  # Check for error
+     # Broadcast message to console
+     cat("(cat) numv = ", numv, "\n")
+     # Return a value
+     paste0("(return) numv = ", numv)
+   }  # end anonymous function
+ )  # end apply
```

Exception Handling in Loops

If the body of the function supplied to the FUN argument is wrapped in `tryCatch()`, then the loop can finish without interruption and return its results.

The messages produced by *errors* and *warnings* can be caught by *handlers* (functions) that are supplied to `tryCatch()`.

The *error* and *warning* messages are bound (passed) to the formal arguments of the *handler* functions that are supplied to `tryCatch()`.

`tryCatch()` always evaluates the expression provided to the `finally` argument, even after an *error* occurs.

```
> # Apply loop with tryCatch
> apply(as.matrix(1:5), 1, function(numv) {  # Anonymous function
+     tryCatch(  # With error handler
+ {  # Body
+   stopifnot(numv != 3)  # Check for error
+   # Broadcast message to console
+   cat("(cat) numv = ", numv, "\t")
+   # Return a value
+   paste0("(return) numv = ", numv)
+ },
+ # Error handler captures error condition
+ error=function(msg)
+   paste0("handler: ", msg),
+ finally=print(paste0("(finally) numv = ", numv))
+     )  # end tryCatch
+   }  # end anonymous function
+ )  # end apply
(cat) numv =  1  [1] "(finally) numv = 1"
(cat) numv =  2  [1] "(finally) numv = 2"
[1] "(finally) numv = 3"
(cat) numv =  4  [1] "(finally) numv = 4"
(cat) numv =  5  [1] "(finally) numv = 5"
[1] "(return) numv = 1"
[2] "(return) numv = 2"
[3] "handler: Error in FUN(newX[, i], ...): numv != 3 is not TRUE\n'
[4] "(return) numv = 4"
[5] "(return) numv = 5"
```

Date Objects

R has a Date class for date objects (but without time).

The function as.Date() parses character strings and coerces numeric objects into Date objects.

R stores Date objects as the number of days since the *epoch* (January 1, 1970).

The function difftime() calculates the difference between Date objects, and returns a time interval object of class difftime.

The "+" and "-" arithmetic operators and the "<" and ">" logical comparison operators are overloaded to allow these operations directly on Date objects.

numeric *year-fraction* dates can be coerced to Date objects using the functions attributes() and structure().

```
> Sys.Date()  # Get today's date
> as.Date(1e3)  # Coerce numeric into date object
> datetime <- as.Date("2014-07-14")  # "%Y-%m-%d" or "%Y/%m/%d"
> datetime
> class(datetime)  # Date object
> as.Date("07-14-2014", "%m-%d-%Y")  # Specify format
> datetime + 20  # Add 20 days
> # Extract internal representation to integer
> as.numeric(datetime)
> datep <- as.Date("07/14/2013", "%m/%d/%Y")
> datep
> # Difference between dates
> difftime(datetime, datep, units="weeks")
> weekdays(datetime)  # Get day of the week
> # Coerce numeric into date-times
> datetime <- 0
> attributes(datetime) <- list(class="Date")
> datetime  # "Date" object
> structure(0, class="Date")  # "Date" object
> structure(10000.25, class="Date")
```

POSIXct Date-time Objects

The POSIXct class in R represents *date-time* objects, that can store both the date and time.

The *clock time* is the time (number of hours, minutes and seconds) in the local *time zone*.

The *moment of time* is the *clock time* in the UTC *time zone*.

POSIXct objects are stored as the number of seconds that have elapsed since the *epoch* (January 1, 1970) in the UTC *time zone*.

POSIXct objects are stored as the *moment of time*, but are printed out as the *clock time* in the local *time zone*.

A *clock time* together with a *time zone* uniquely specifies a *moment of time*.

The function as.POSIXct() can parse a character string (representing the *clock time*) and a *time zone* into a POSIXct object.

POSIX is an acronym for "Portable Operating System Interface".

```
> datetime <- Sys.time()  # Get today's date and time
> datetime
> class(datetime)  # POSIXct object
> # POSIXct stored as integer moment of time
> as.numeric(datetime)
> # Parse character string "%Y-%m-%d %H:%M:%S" to POSIXct object
> datetime <- as.POSIXct("2014-07-14 13:30:10")
> # Different time zones can have same clock time
> as.POSIXct("2014-07-14 13:30:10", tz="America/New_York")
> as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # Format argument allows parsing different date-time string format
> as.POSIXct("07/14/2014 13:30:10", format="%m/%d/%Y %H:%M:%S",
+      tz="America/New_York")
```

Operations on POSIXct Objects

The "+" and "-" arithmetic operators are overloaded to allow addition and subtraction operations on POSIXct objects.

The "<" and ">" logical comparison operators are also overloaded to allow direct comparisons between POSIXct objects.

Operations on POSIXct objects are equivalent to the same operations on the internal integer representation of POSIXct (number of seconds since the *epoch*).

Subtracting POSIXct objects creates a time interval object of class difftime.

The method seq.POSIXt creates a vector of POSIXct *date-times*.

```
> # Same moment of time corresponds to different clock times
> timeny <- as.POSIXct("2014-07-14 13:30:10", tz="America/New_York")
> timeldn <- as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # Add five hours to POSIXct
> timeny + 5*60*60
> # Subtract POSIXct
> timeny - timeldn
> class(timeny - timeldn)
> # Compare POSIXct
> timeny > timeldn
> # Create vector of POSIXct times during trading hours
> timev <- seq(
+   from=as.POSIXct("2014-07-14 09:30:00", tz="America/New_York"),
+   to=as.POSIXct("2014-07-14 16:00:00", tz="America/New_York"),
+   by="10 min")
> head(timev, 3)
> tail(timev, 3)
```

Moment of Time and Clock Time

`as.POSIXct()` can also coerce `integer` objects into `POSIXct`, given an `origin` in time.

The same *moment of time* corresponds to different *clock times* in different *time zones*.

The same *clock times* in different *time zones* correspond to different *moments of time*.

```
> # POSIXct is stored as integer moment of time
> datetimen <- as.numeric(datetime)
> # Same moment of time corresponds to different clock times
> as.POSIXct(datetimen, origin="1970-01-01", tz="America/New_York")
> as.POSIXct(datetimen, origin="1970-01-01", tz="UTC")
> # Same clock time corresponds to different moments of time
> as.POSIXct("2014-07-14 13:30:10", tz="America/New_York") -
+   as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # Add 20 seconds to POSIXct
> datetime + 20
```

Methods for Manipulating POSIXct Objects

The generic function `format()` formats `R` objects for printing and display.

The method `format.POSIXct()` parses `POSIXct` objects into a `character` string representing the *clock time* in a given *time zone*.

The method `as.POSIXct.Date()` parses `Date` objects into `POSIXct`, and assigns to them the *moment of time* corresponding to midnight `UTC`.

`POSIX` is an acronym for "Portable Operating System Interface".

```
> datetime  # POSIXct date and time
> # Parse POSIXct to string representing the clock time
> format(datetime)
> class(format(datetime))  # Character string
> # Get clock times in different time zones
> format(datetime, tz="America/New_York")
> format(datetime, tz="UTC")
> # Format with custom format strings
> format(datetime, "%m/%Y")
> format(datetime, "%m-%d-%Y %H hours")
> # Trunc to hour
> format(datetime, "%m-%d-%Y %H:00:00")
> # Date converted to midnight UTC moment of time
> as.POSIXct(Sys.Date())
> as.POSIXct(as.numeric(as.POSIXct(Sys.Date())),
+      origin="1970-01-01",
+      tz="UTC")
```

POSIXlt Date-time Objects

The POSIXlt class in R represents *date-time* objects, that are stored internally as a list.

The function as.POSIXlt() can parse a character string (representing the *clock time*) and a *time zone* into a POSIXlt object.

The method format.POSIXlt() parses POSIXlt objects into a character string representing the *clock time* in a given *time zone*.

The function as.POSIXlt() can also parse a POSIXct object into a POSIXlt object, and as.POSIXct() can perform the reverse.

Adding a number to POSIXlt causes implicit coercion to POSIXct.

POSIXct and POSIXlt are two derived classes from the POSIXt class.

The methods round.POSIXt() and trunc.POSIXt() round and truncate POSIXt objects, and return POSIXlt objects.

```
> # Parse character string "%Y-%m-%d %H:%M:%S" to POSIXlt object
> datetime <- as.POSIXlt("2014-07-14 18:30:10")
> datetime
> class(datetime)  # POSIXlt object
> as.POSIXct(datetime)  # Coerce to POSIXct object
> # Extract internal list representation to vector
> unlist(datetime)
> datetime + 20  # Add 20 seconds
> class(datetime + 20)  # Implicit coercion to POSIXct
> trunc(datetime, units="hours")  # Truncate to closest hour
> trunc(datetime, units="days")  # Truncate to closest day
> methods(trunc)  # Trunc methods
> trunc.POSIXt
```

Time Zones and Date-time Conversion

date-time objects require a *time zone* to be uniquely specified.

UTC stands for "Universal Time Coordinated", and is synonymous with GMT, but doesn't change with Daylight Saving Time.

EST stands for "Eastern Standard Time", and is UTC - 5 hours.

EDT stands for "Eastern Daylight Time", and is UTC - 4 hours.

The function Sys.setenv() can be used to set the default *time zone*, but the environment variable "TZ" must be capitalized.

```
> # Set time-zone to UTC
> Sys.setenv(TZ="UTC")
> Sys.timezone()  # Get time-zone
> Sys.time()  # Today's date and time
> # Set time-zone back to New York
> Sys.setenv(TZ="America/New_York")
> Sys.time()  # Today's date and time
> # Standard Time in effect
> as.POSIXct("2013-03-09 11:00:00", tz="America/New_York")
> # Daylight Savings Time in effect
> as.POSIXct("2013-03-10 11:00:00", tz="America/New_York")
> datetime <- Sys.time()  # Today's date and time
> # Convert to character in different TZ
> format(datetime, tz="America/New_York")
> format(datetime, tz="UTC")
> # Parse back to POSIXct
> as.POSIXct(format(datetime, tz="America/New_York"))
> # Difference between New_York time and UTC
> as.POSIXct(format(Sys.time(), tz="UTC")) -
+   as.POSIXct(format(Sys.time(), tz="America/New_York"))
```

Manipulating Date-time Objects Using *lubridate*

The package *lubridate* contains functions for manipulating POSIXct date-time objects.

The ymd(), dmy(), etc. functions parse character and numeric *year-fraction* dates into POSIXct objects.

The mday(), month(), year(), etc. accessor functions extract date-time components.

The function decimal_date() converts POSIXct objects into numeric *year-fraction* dates.

The function date_decimal() converts numeric *year-fraction* dates into POSIXct objects.

```
> library(lubridate)  # Load lubridate
> # Parse strings into date-times
> as.POSIXct("07-14-2014", format="%m-%d-%Y", tz="America/New_York")
> datetime <- lubridate::mdy("07-14-2014", tz="America/New_York")
> datetime
> class(datetime)  # POSIXct object
> lubridate::dmy("14.07.2014", tz="America/New_York")
>
> # Parse numeric into date-times
> as.POSIXct(as.character(14072014), format="%d%m%Y",
+                 tz="America/New_York")
> lubridate::dmy(14072014, tz="America/New_York")
>
> # Parse decimal to date-times
> lubridate::decimal_date(datetime)
> lubridate::date_decimal(2014.25, tz="America/New_York")
> date_decimal(decimal_date(datetime), tz="America/New_York")
```

Time Zones Using *lubridate*

The package *lubridate* simplifies *time zone* calculations.

The package *lubridate* uses the *UTC time zone* as default.

The function `with_tz()` creates a date-time object with the same moment of time in a different *time zone*.

The function `force_tz()` creates a date-time object with the same clock time in a different *time zone*.

```
> datetime <- lubridate::ymd_hms(20140714142010,
+                tz="America/New_York")
> datetime
> # Get same moment of time in "UTC" time zone
> lubridate::with_tz(datetime, "UTC")
> as.POSIXct(format(datetime, tz="UTC"), tz="UTC")
> # Get same clock time in "UTC" time zone
> lubridate::force_tz(datetime, "UTC")
> as.POSIXct(format(datetime, tz="America/New_York"),
+      tz="UTC")
> # Same moment of time
> datetime - with_tz(datetime, "UTC")
> # Different moments of time
> datetime - force_tz(datetime, "UTC")
```

lubridate Time Span Objects

lubridate has two time span classes: `durations` and `periods`.

`durations` specify exact time spans, such as numbers of seconds, hours, days, etc.

The functions `ddays()`, `dyears()`, etc. return `duration` objects.

`periods` specify relative time spans that don't have a fixed length, such as months, years, etc.

`periods` account for variable days in the months, for Daylight Savings Time, and for leap years.

The functions `days()`, `months()`, `years()`, etc. return `period` objects.

```
> # Daylight Savings Time handling periods vs durations
> datetime <- as.POSIXct("2013-03-09 11:00:00", tz="America/New_York
> datetime
> datetime + lubridate::ddays(1)  # Add duration
> datetime + lubridate::days(1)  # Add period
>
> leap_year(2012)  # Leap year
> datetime <- lubridate::dmy(01012012, tz="America/New_York")
> datetime
> datetime + lubridate::dyears(1)  # Add duration
> datetime + lubridate::years(1)  # Add period
```

Adding Time Spans to Date-time Objects

periods allow calculating future dates with the same day of the month, or month of the year.

```
> datetime <- lubridate::ymd_hms(20140714142010, tz="America/New_Yor
> datetime
> # Add periods to a date-time
> c(datetime + lubridate::seconds(1), datetime + lubridate::minutes(
+   datetime + lubridate::days(1), datetime + period(months=1))
>
> # Create vectors of dates
> datetime <- lubridate::ymd(20140714, tz="America/New_York")
> datetime + 0:2 * period(months=1)  # Monthly dates
> datetime + period(months=0:2)
> datetime + 0:2 * period(months=2)  # bi-monthly dates
> datetime + seq(0, 5, by=2) * period(months=1)
> seq(datetime, length=3, by="2 months")
```

End-of-month Dates

Adding monthly periods can create invalid dates.

The operators %m+% and %m-% add or subtract monthly periods to account for the varible number of days per month.

This allows creating vectors of end-of-month dates.

```
> # Adding monthly periods can create invalid dates
> datetime <- lubridate::ymd(20120131, tz="America/New_York")
> datetime + 0:2 * period(months=1)
> datetime + period(months=1)
> datetime + period(months=2)
> # Create vector of end-of-month dates
> datetime %m-% months(13:1)
```

Package *RQuantLib* Calendar Functions

The package *RQuantLib* is an interface to the *QuantLib* open source C/C++ library for quantitative finance, mostly designed for pricing fixed-income instruments and options.

The *QuantLib* library also contains calendar functions for determining holidays and business days in many different jurisdictions.

```
> library(RQuantLib)  # Load RQuantLib
>
> # Create daily date series of class "Date"
> datev <- Sys.Date() + -5:2
> datev
[1] "2024-10-10" "2024-10-11" "2024-10-12" "2024-10-13" "2024-10-14"
[6] "2024-10-15" "2024-10-16" "2024-10-17"
>
> # Create Boolean vector of business days
> # Use RQuantLib calendar
> isbusday <- RQuantLib::isBusinessDay(
+   calendar="UnitedStates/GovernmentBond", datev)
>
> # Create daily series of business days
> datev[isbusday]
[1] "2024-10-10" "2024-10-11" "2024-10-15" "2024-10-16" "2024-10-17"
```

Review of Date-time Classes in R

The `Date` class from the `base` package is suitable for *daily* time series.

The `POSIXct` class from the `base` package is suitable for *intra-day* time series.

The `yearmon` and `yearqtr` classes from the *zoo* package are suitable for *quarterly* and *monthly* time series.

```
> datetime <- Sys.Date()  # Create date series of class "Date"
> datev <- datetime + 0:365  # Daily series over one year
> head(datev, 4)  # Print first few dates
[1] "2024-10-15" "2024-10-16" "2024-10-17" "2024-10-18"
> format(head(datev, 4), "%m/%d/%Y")  # Print first few dates
[1] "10/15/2024" "10/16/2024" "10/17/2024" "10/18/2024"
> # Create daily date-time series of class "POSIXct"
> datev <- seq(Sys.time(), by="days", length.out=365)
> head(datev, 4)  # Print first few dates
[1] "2024-10-15 18:42:37 EDT" "2024-10-16 18:42:37 EDT"
[3] "2024-10-17 18:42:37 EDT" "2024-10-18 18:42:37 EDT"
> format(head(datev, 4), "%m/%d/%Y %H:%M:%S")  # Print first few dat
[1] "10/15/2024 18:42:37" "10/16/2024 18:42:37" "10/17/2024 18:42:37
[4] "10/18/2024 18:42:37"
> # Create series of monthly dates of class "zoo"
> monthv <- yearmon(2010+0:36/12)
> head(monthv, 4)  # Print first few dates
[1] "Jan 2010" "Feb 2010" "Mar 2010" "Apr 2010"
> # Create series of quarterly dates of class "zoo"
> qrtv <- yearqtr(2010+0:16/4)
> head(qrtv, 4)  # Print first few dates
[1] "2010 Q1" "2010 Q2" "2010 Q3" "2010 Q4"
> # Parse quarterly "zoo" dates to POSIXct
> Sys.setenv(TZ="UTC")
> as.POSIXct(head(qrtv, 4))
[1] "2010-01-01 UTC" "2010-04-01 UTC" "2010-07-01 UTC" "2010-10-01 U
```

Time Series Objects of Class *ts*

Time series are data objects that contain a *date-time* index and data associated with it.

The native time series class in `R` is *ts*.

ts time series are *regular*, i.e. they can only have an equally spaced *date-time* index.

ts time series have a `numeric` *date-time* index, usually encoded as a *year-fraction*, or some other unit, like number of months, etc.

For example the date `"2015-03-31"` can be encoded as a *year-fraction* equal to `2015.244`.

The *stats* base package contains functions for manipulating time series objects of class *ts*.

The function `ts()` creates a *ts* time series from a `numeric` vector or matrix, and from the associated *date-time* information (the number of data per time unit: year, month, etc.).

The `frequency` argument is the number of observations per unit of time.

For example, if the *date-time* index is encoded as a *year-fraction*, then `frequency=12` means 12 monthly data points per year.

```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Create daily time series ending today
> startd <- decimal_date(Sys.Date()-6)
> endd <- decimal_date(Sys.Date())
> # Create vector of geometric Brownian motion
> datav <- exp(cumsum(rnorm(6)/100))
> tstep <- NROW(datav)/(endd-startd)
> timeser <- ts(data=datav, start=startd, frequency=tstep)
> timeser  # Display time series
> # Display index dates
> as.Date(date_decimal(zoo::coredata(time(timeser))))
> # bi-monthly geometric Brownian motion starting mid-1990
> timeser <- ts(data=exp(cumsum(rnorm(96)/100)),
+        frequency=6, start=1990.5)
```

Manipulating *ts* Time Series

ts time series don't store their *date-time* indices, and instead store only a `"tsp"` attribute that specifies the index `start` and `end` dates and its `frequency`.

The *date-time* index is calculated as needed from the `"tsp"` attribute.

The function `time()` extracts the *date-time* index of a *ts* time series object.

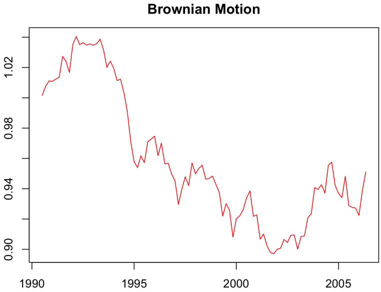
The function `window()` subsets the a *ts* time series object.

```
> # Show some methods for class "ts"
> matrix(methods(class="ts")[3:8], ncol=2)
> # "tsp" attribute specifies the date-time index
> attributes(timeser)
> # Extract the index
> tail(time(timeser), 11)
> # The index is equally spaced
> diff(tail(time(timeser), 11))
> # Subset the time series
> window(timeser, start=1992, end=1992.25)
```

Plotting *ts* Time Series Objects

The method `plot.ts()` plots *ts* time series objects.

```
> # Create plot
> plot(timeser, type="l", col="red", lty="solid",
+      xlab="", ylab="")
> title(main="Brownian Motion", line=1)  # Add title
```



EuStockMarkets Data

R includes a number of base packages that are already installed and loaded.

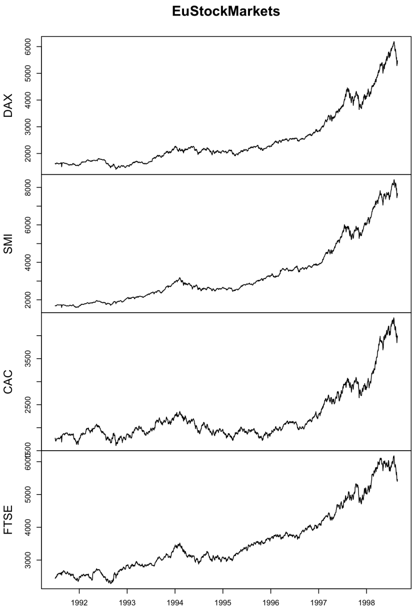
datasets is a base package containing various datasets, for example: EuStockMarkets.

The EuStockMarkets dataset contains daily closing prices of european stock indices.

EuStockMarkets is a mts() time series object.

The EuStockMarkets *date-time* index is equally spaced (*regular*), so the *year-fraction* dates don't correspond to actual trading days.

```
> class(EuStockMarkets)  # Multiple ts object
> dim(EuStockMarkets)
> head(EuStockMarkets, 3)  # Get first three rows
> # EuStockMarkets index is equally spaced
> diff(tail(time(EuStockMarkets), 11))
> # Plot all the columns in separate panels
> plot(EuStockMarkets, main="EuStockMarkets", xlab="")
```

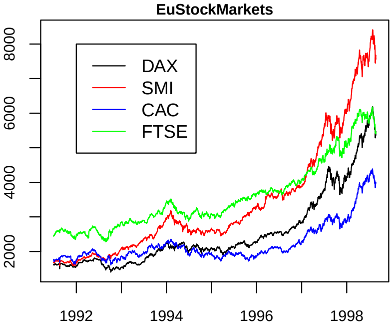


Plotting EuStockMarkets Data

The argument `plot.type="single"` for method `plot.zoo()` allows plotting multiple lines in a single panel (pane).

The four `EuStockMarkets` time series can be plotted in a single panel (pane).

```
> # Plot in single panel
> plot(EuStockMarkets, main="EuStockMarkets",
+      xlab="", ylab="", plot.type="single",
+      col=c("black", "red", "blue", "green"))
> # Add legend
> legend(x=1992, y=8000,
+  legend=colnames(EuStockMarkets),
+  col=c("black", "red", "blue", "green"),
+  lwd=6, lty=1)
```



zoo Time Series Objects

The package *zoo* is designed for managing *irregular* time series and ordered objects of class *zoo*.

Irregular time series have *date-time* indices that aren't equally spaced (because of weekends, overnight hours, etc.).

The function `zoo()` creates a *zoo* object from a `numeric` vector or matrix, and an associated *date-time* index.

The *zoo* index is a vector of *date-time* objects, and can be from any *date-time* class.

The *zoo* class can manage *irregular* time series whose *date-time* index isn't equally spaced.

```
> library(zoo)  # Load package zoo
> # Create zoo time series of random returns
> datev <- Sys.Date() + 0:11
> zoots <- zoo(rnorm(NROW(datev)), order.by=datev)
> zoots
2024-10-15 2024-10-16 2024-10-17 2024-10-18 2024-10-19 2024-10-20 20
    0.1450     0.4383     0.1532     1.0849     1.9995    -0.8119
2024-10-22 2024-10-23 2024-10-24 2024-10-25 2024-10-26
    0.5859     0.3601    -0.0253     0.1509     0.1101
> attributes(zoots)
$index
 [1] "2024-10-15" "2024-10-16" "2024-10-17" "2024-10-18" "2024-10-19
 [6] "2024-10-20" "2024-10-21" "2024-10-22" "2024-10-23" "2024-10-24
[11] "2024-10-25" "2024-10-26"

$class
[1] "zoo"
> class(zoots)  # Class "zoo"
[1] "zoo"
> tail(zoots, 3)  # Get last few elements
2024-10-24 2024-10-25 2024-10-26
   -0.0253     0.1509     0.1101
```

Operations on *zoo* Time Series

The function `zoo::coredata()` extracts the data contained in *zoo* object, and returns a vector or matrix.

The function `zoo::index()` extracts the time index of a *zoo* object.

The function `xts::.index()` extracts the time index expressed in the number of seconds.

The functions `start()` and `end()` return the time index values of the first and last elements of a *zoo* object.

The functions `cumsum()`, `cummax()`, and `cummin()` return cumulative sums, minima and maxima of a *zoo* object.

```
> zoo::coredata(zoots)  # Extract coredata
 [1]  0.1450  0.4383  0.1532  1.0849  1.9995 -0.8119  0.1603  0.5859
[10] -0.0253  0.1509  0.1101
> zoo::index(zoots)  # Extract time index
 [1] "2024-10-15" "2024-10-16" "2024-10-17" "2024-10-18" "2024-10-19
 [6] "2024-10-20" "2024-10-21" "2024-10-22" "2024-10-23" "2024-10-24
[11] "2024-10-25" "2024-10-26"
> start(zoots)  # First date
[1] "2024-10-15"
> end(zoots)  # Last date
[1] "2024-10-26"
> zoots[start(zoots)]  # First element
2024-10-15
     0.145
> zoots[end(zoots)]  # Last element
2024-10-26
      0.11
> zoo::coredata(zoots) <- rep(1, NROW(zoots))  # Replace coredata
> cumsum(zoots)  # Cumulative sum
2024-10-15 2024-10-16 2024-10-17 2024-10-18 2024-10-19 2024-10-20 20
         1          2          3          4          5          6
2024-10-22 2024-10-23 2024-10-24 2024-10-25 2024-10-26
         8          9         10         11         12
> cummax(cumsum(zoots))
2024-10-15 2024-10-16 2024-10-17 2024-10-18 2024-10-19 2024-10-20 20
         1          2          3          4          5          6
2024-10-22 2024-10-23 2024-10-24 2024-10-25 2024-10-26
         8          9         10         11         12
> cummin(cumsum(zoots))
2024-10-15 2024-10-16 2024-10-17 2024-10-18 2024-10-19 2024-10-20 20
         1          1          1          1          1          1
2024-10-22 2024-10-23 2024-10-24 2024-10-25 2024-10-26
         1          1          1          1          1
```

Single Column *zoo* Time Series

Single column *zoo* time series usually don't have a dimension attribute (they have a NULL dimension), and they don't have a column name, unlike multi-column *zoo* time series.

Single column *zoo* time series without a dimension attribute should be avoided, since they can cause hard to detect bugs.

If a single column *zoo* time series is created from a single column matrices, then it have a dimension attribute, and can be assigned a column name.

```
> zoots <- zoo(matrix(cumsum(rnorm(10)), nc=1),
+   order.by=seq(from=as.Date("2013-06-15"), by="day", len=10))
> colnames(zoots) <- "zoots"
> tail(zoots)
            zoots
2013-06-19  2.63
2013-06-20  2.11
2013-06-21  2.24
2013-06-22  2.08
2013-06-23  2.15
2013-06-24  2.08
> dim(zoots)
[1] 10  1
> attributes(zoots)
$dim
[1] 10  1

$index
 [1] "2013-06-15" "2013-06-16" "2013-06-17" "2013-06-18" "2013-06-19
 [6] "2013-06-20" "2013-06-21" "2013-06-22" "2013-06-23" "2013-06-24

$class
[1] "zoo"

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "zoots"
```

The lag() and diff() Functions

The method `lag.zoo()` returns a lagged version of a *zoo* time series, shifting the time index by `"k"` observations.

If `"k"` is positive, then `lag.zoo()` shifts values from the future to the present, and if `"k"` is negative then it shifts them from the past.

This is the opposite of what is usually considered as a positive *lag*.

A positive *lag* should replace the current value with values from the past (negative lags should replace with values from the future).

The method `diff.zoo()` returns the difference between a *zoo* time series and its proper lagged version from the past, given a positive *lag* value.

By default, the methods `lag.zoo()` and `diff.zoo()` omit any `NA` values they may have produced, and return shorter time series.

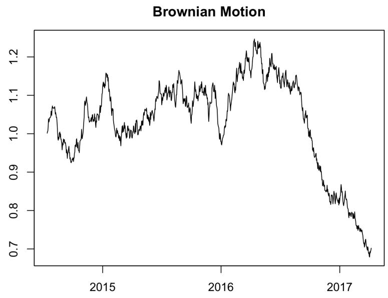
If the `"na.pad"` argument is set to `TRUE`, then they return time series of the same length, with `NA` values added where needed.

```
> zoo::coredata(zoots) <- (1:10)^2  # Replace coredata
> zoots
           zoots
2013-06-15     1
2013-06-16     4
2013-06-17     9
2013-06-18    16
2013-06-19    25
2013-06-20    36
2013-06-21    49
2013-06-22    64
2013-06-23    81
2013-06-24   100
> lag(zoots)  # One day lag
           zoots
2013-06-15     4
2013-06-16     9
2013-06-17    16
2013-06-18    25
2013-06-19    36
2013-06-20    49
2013-06-21    64
2013-06-22    81
2013-06-23   100
> lag(zoots, 2)  # Two day lag
           zoots
2013-06-15     9
2013-06-16    16
2013-06-17    25
2013-06-18    36
2013-06-19    49
2013-06-20    64
2013-06-21    81
2013-06-22   100
> lag(zoots, k=-1)  # Proper one day lag
           zoots
2013-06-16     1
```

Plotting *zoo* Time Series

zoo time series can be plotted using the generic function `plot()`, which dispatches the `plot.zoo()` method.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> library(zoo)  # Load package zoo
> # Create index of daily dates
> datev <- seq(from=as.Date("2014-07-14"), by="day", length.out=100(
> # Create vector of geometric Brownian motion
> datav <- exp(cumsum(rnorm(NROW(datev))/100))
> # Create zoo series of geometric Brownian motion
> zoots <- zoo(x=datav, order.by=datev)
> # Plot using method plot.zoo()
> plot.zoo(zoots, xlab="", ylab="")
> title(main="Brownian Motion", line=1)  # Add title
```



Subsetting *zoo* Time Series

zoo time series can be subset in similar ways to `matrices` and *ts* time series.

The function `window()` can also subset *zoo* time series objects.

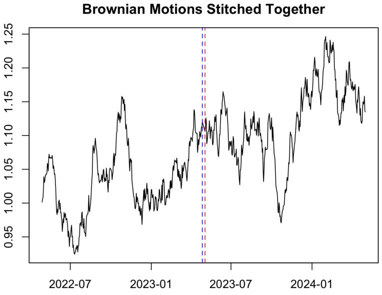
In addition, *zoo* time series can be subset using `Date` objects.

```
> # Subset zoo as matrix
> zoots[459:463, 1]
> # Subset zoo using window()
> window(zoots,
+  start=as.Date("2014-10-15"),
+  end=as.Date("2014-10-19"))
> # Subset zoo using Date object
> zoots[as.Date("2014-10-15")]
```

Sequential Joining *zoo* Time Series

The *zoo* time series can be joined sequentially using function `rbind()`.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> library(zoo)  # Load package zoo
> # Create daily date series of class "Date"
> tday <- Sys.Date()
> index1 <- seq(tday-2*365, by="days", length.out=365)
> # Create zoo time series of random returns
> zoo1 <- zoo(rnorm(NROW(index1)), order.by=index1)
> # Create another zoo time series of random returns
> index2 <- seq(tday-360, by="days", length.out=365)
> zoo2 <- zoo(rnorm(NROW(index2)), order.by=index2)
> # rbind the two time series - ts1 supersedes ts2
> zooub2 <- zoo2[zoo::index(zoo2) > end(zoo1)]
> zoo3 <- rbind(zoo1, zooub2)
> # Plot zoo time series of geometric Brownian motion
> plot(exp(cumsum(zoo3)/100), xlab="", ylab="")
> # Add vertical lines at stitch point
> abline(v=end(zoo1), col="blue", lty="dashed")
> abline(v=start(zoo2), col="red", lty="dashed")
> title(main="Brownian Motions Stitched Together", line=1)  # Add title
```



Merging *zoo* Time Series

zoo time series can be combined concurrently by joining their columns using function `merge()`.

Function `merge()` is similar to function `cbind()`.

If the `all=TRUE` option is set, then `merge()` returns the union of their dates, otherwise it returns their intersection.

The `merge()` operation can produce `NA` values.

```
> # Create daily date series of class "Date"
> index1 <- Sys.Date() + -3:1
> # Create zoo time series of random returns
> zoo1 <- zoo(rnorm(NROW(index1)), order.by=index1)
> # Create another zoo time series of random returns
> index2 <- Sys.Date() + -1:3
> zoo2 <- zoo(rnorm(NROW(index2)), order.by=index2)
> merge(zoo1, zoo2)  # union of dates
> # Intersection of dates
> merge(zoo1, zoo2, all=FALSE)
```

Managing NA Values

Binding two time series that don't share the same time index produces NA values.

There are two dedicated functions for managing NA values in time series:

- `stats::na.omit()` removes whole rows of data containing NA values.
- `zoo::na.locf()` replaces NA values with the most recent non-NA values prior to it (*locf* stands for *last observation carry forward*).

Copying the last non-NA values forward causes less data loss than removing whole rows of data.

`na.locf()` with argument `fromLast=TRUE` operates in reverse order, starting from the end.

But copying values forward requires initializing the first row of data, to guarantee that initial NA values are also over-written.

The initial NA *prices* can be initialized to the first non-NA price in the future, which can be done by calling `zoo::na.locf()` with the argument `fromLast=TRUE`.

But the initial NA values in *returns* data should be initialized to *zero*, without carrying data backward from the future, to avoid data *snooping*.

```
> # Create matrix containing NA values
> matv <- sample(18)
> matv[sample(NROW(matv), 4)] <- NA
> matv <- matrix(matv, nc=3)
> # Replace NA values with most recent non-NA values
> zoo::na.locf(matv)
> # Get time series of prices
> pricev <- mget(c("VTI", "VXX"), envir=rutils::etfenv)
> pricev <- lapply(pricev, quantmod::Cl)
> pricev <- rutils::do_call(cbind, pricev)
> sum(is.na(pricev))
> # Carry forward and backward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> pricev <- zoo::na.locf(pricev, na.rm=FALSE, fromLast=TRUE)
> sum(is.na(pricev))
> # Remove whole rows containing NA returns
> retp <- rutils::etfenv$returns
> sum(is.na(retp))
> retp <- na.omit(retp)
> # Or carry forward non-NA returns (preferred)
> retp <- rutils::etfenv$returns
> retp[1, is.na(retp[1, ])] <- 0
> retp <- zoo::na.locf(retp, na.rm=FALSE)
> sum(is.na(retp))
```

Managing NA Values in "xts" Time Series

The function na.locf.xts() from package *xts* is faster than zoo::na.locf(), but it only operates on time series of class "xts".

```
> # Replace NAs in xts time series
> pricev <- rutils::etfenv$prices[, 1]
> head(pricev)
> sum(is.na(pricev))
> library(quantmod)
> pricezoo <- zoo::na.locf(pricev, na.rm=FALSE, fromLast=TRUE)
> pricexts <- xts:::na.locf.xts(pricev, fromLast=TRUE)
> all.equal(pricezoo, pricexts, check.attributes=FALSE)
> library(microbenchmark)
> summary(microbenchmark(
+   zoo=zoo::na.locf(pricev, fromLast=TRUE),
+   xts=xts:::na.locf.xts(pricev, fromLast=TRUE),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing Time Series Objects Into *zoo*

The generic function `as.zoo()` coerces objects into *zoo* time series.

The function `as.zoo()` creates a *zoo* object with a `numeric` *date-time* index, with *date-time* encoded as a *year-fraction*.

The *year-fraction* can be *approximately* converted to a `Date` object by first calculating the number of days since the *epoch* (1970), and then coercing the `numeric` days using `as.Date()`.

The function `date_decimal()` from package *lubridate* converts `numeric` *year-fraction* dates into `POSIXct` objects.

The function `date_decimal()` provides a more accurate way of converting a *year-fraction* index to `POSIXct`.

```
> class(EuStockMarkets)  # Multiple ts object
> # Coerce mts object into zoo
> zoots <- as.zoo(EuStockMarkets)
> class(zoo::index(zoots))  # Index is numeric
> head(zoots, 3)
> # Approximately convert index into class "Date"
> zoo::index(zoots) <-
+   as.Date(365*(zoo::index(zoots)-1970))
> head(zoots, 3)
> # Convert index into class "POSIXct"
> zoots <- as.zoo(EuStockMarkets)
> zoo::index(zoots) <- date_decimal(zoo::index(zoots))
> head(zoots, 3)
```

Coercing *zoo* Time Series Into Class *ts*

The generic function `as.ts()` from package *stats* coerces time series objects (including *zoo*) into *ts* time series.

The function `as.ts()` creates a *ts* object with a `frequency=1`, implying a *"day"* time unit, instead of a *"year"* time unit suitable for *year-fraction* dates.

A *ts* time series can be created from a *zoo* using the function `ts()`, after extracting the data and date attributes from *zoo*.

The function `decimal_date()` from package *lubridate* converts `POSIXct` objects into `numeric` *year-fraction* dates.

```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Create index of daily dates
> datev <- seq(from=as.Date("2014-07-14"), by="day", length.out=100
> # Create vector of geometric Brownian motion
> datav <- exp(cumsum(rnorm(NROW(datev))/100))
> # Create zoo time series of geometric Brownian motion
> zoots <- zoo(x=datav, order.by=datev)
> head(zoots, 3)  # zoo object
> # as.ts() creates ts object with frequency=1
> timeser <- as.ts(zoots)
> tsp(timeser)  # Frequency=1
> # Get start and end dates of zoots
> startd <- decimal_date(start(zoots))
> endd <- decimal_date(end(zoots))
> # Calculate frequency of zoots
> tstep <- NROW(zoots)/(endd-startd)
> datav <- zoo::coredata(zoots)  # Extract data from zoots
> # Create ts object using ts()
> timeser <- ts(data=datav, start=startd, frequency=tstep)
> # Display start of time series
> window(timeser, start=start(timeser), end=start(timeser)+4/365)
> head(time(timeser))  # Display index dates
> head(as.Date(date_decimal(zoo::coredata(time(timeser)))))
```

Coercing Irregular Time Series Into Class *ts*

Irregular time series cannot be properly coerced into *ts* time series without modifying their index.

The function `as.ts()` creates `NA` values when it coerces irregular time series into a *ts* time series.

```
> # Create weekday Boolean vector
> wkdays <- weekdays(zoo::index(zoots))
> wkdayl <- !((wkdays == "Saturday") | (wkdays == "Sunday"))
> # Remove weekends from zoo time series
> zoots <- zoots[wkdayl, ]
> head(zoots, 7)  # zoo object
> # as.ts() creates NA values
> timeser <- as.ts(zoots)
> head(timeser, 7)
> # Create vector of regular dates, including weekends
> datev <- seq(from=start(zoots), by="day", length.out=NROW(zoots))
> zoo::index(zoots) <- datev
> timeser <- as.ts(zoots)
> head(timeser, 7)
```

Class *xts* Time Series Objects

The package *xts* defines time series objects of class *xts*,

- Class *xts* is an extension of the *zoo* class (derived from *zoo*),
- Class *xts* is the most widely accepted time series class,
- Class *xts* is designed for high-frequency and *OHLC* data,
- Class *xts* contains many convenient functions for plotting, calculating rolling max, min, etc.

The function `xts()` creates a *xts* object from a `numeric` vector or matrix, and an associated *date-time* index.

The *xts* index is a vector of *date-time* objects, and can be from any *date-time* class.

The *xts* class can manage *irregular* time series whose *date-time* index isn't equally spaced.

```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> library(xts)  # Load package xts
> # Create xts time series of random returns
> datev <- Sys.Date() + 0:3
> xtsv <- xts(rnorm(NROW(datev)), order.by=datev)
> names(xtsv) <- "random"
> xtsv
> tail(xtsv, 3)  # Get last few elements
> first(xtsv)  # Get first element
> last(xtsv)  # Get last element
> class(xtsv)  # Class "xts"
> attributes(xtsv)
> # Get the time zone of an xts object
> tzone(xtsv)
```

Coercing *zoo* Time Series Into Class *xts*

The function as.xts() coerces *zoo* time series into *xts* series.

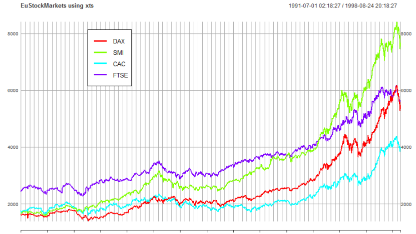
as.xts() preserves the *index* attributes of the original time series.

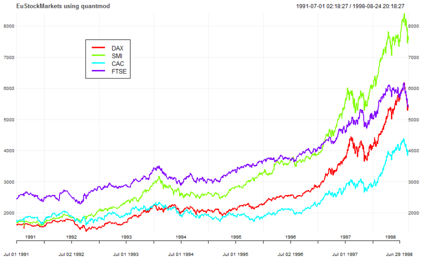
xts can be plotted using the generic function plot(), which dispatches the plot.xts() method.

```
> load(file="/Users/jerzy/Develop/lecture_slides/data/zoo_data.RData
> class(zoo_stx)
> # as.xts() coerces zoo series into xts series
> library(xts)  # Load package xts
> pricexts <- as.xts(zoo_stx)
> dim(pricexts)
> head(pricexts[, 1:4], 4)
> # Plot using plot.xts method
> xts::plot.xts(pricexts[, "Close"], xlab="", ylab="", main="")
> title(main="Stock Prices")  # Add title
```

Plotting Multiple *xts* Using Packages *xts* and *quantmod*

```
> library(lubridate)  # Load lubridate
> # Coerce EuStockMarkets into class xts
> xtsv <- xts(zoo::coredata(EuStockMarkets),
+       order.by=date_decimal(zoo::index(EuStockMarkets)))
> # Plot all columns in single panel: xts v.0.9-8
> colorv <- rainbow(NCOL(xtsv))
> plot(xtsv, main="EuStockMarkets using xts",
+      col=colorv, major.ticks="years",
+      minor.ticks=FALSE)
> legend("topleft", legend=colnames(EuStockMarkets),
+  inset=0.2, cex=0.7, , lty=rep(1, NCOL(xtsv)),
+  lwd=3, col=colorv, bg="white")
> # Plot only first column: xts v.0.9-7
> plot(xtsv[, 1], main="EuStockMarkets using xts",
+      col=colorv[1], major.ticks="years",
+      minor.ticks=FALSE)
> # Plot remaining columns
> for (colnum in 2:NCOL(xtsv))
+   lines(xtsv[, colnum], col=colorv[colnum])
> # Plot using quantmod
> library(quantmod)
> plotheme <- chart_theme()
> plotheme$col$line.col <- colors
> chart_Series(x=xtsv, theme=plotheme,
+        name="EuStockMarkets using quantmod")
> legend("topleft", legend=colnames(EuStockMarkets),
+  inset=0.2, cex=0.7, , lty=rep(1, NCOL(xtsv)),
+  lwd=3, col=colorv, bg="white")
```





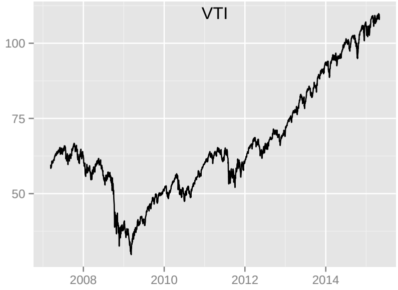
Plotting *xts* Using Package *ggplot2*

xts time series can be plotted using the package *ggplot2*.

The function `qplot()` is the simplest function in the *ggplot2* package, and allows creating line and bar plots.

The function `theme()` customizes plot objects.

```
> library(ggplot2)
> pricev <- rutils::etfenv$prices[, 1]
> pricev <- na.omit(pricev)
> # Create ggplot object
> plotobj <- qplot(x=zoo::index(pricev),
+           y=as.numeric(pricev),
+           geom="line",
+           main=names(pricev)) +
+   xlab("") + ylab("") +
+   theme(  # Add legend and title
+     legend.position=c(0.1, 0.5),
+     plot.title=element_text(vjust=-2.0),
+     plot.background=element_blank()
+   )  # end theme
> # Render ggplot object
> plotobj
```

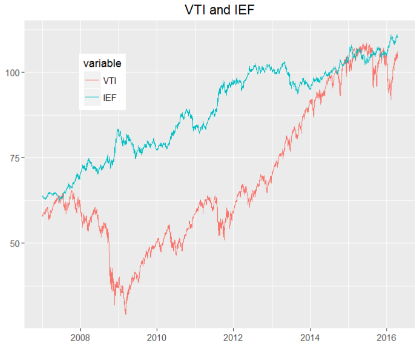


Plotting Multiple *xts* Using Package *ggplot2*

Multiple *xts* time series can be plotted using the function `ggplot()` from package *ggplot2*.

But *ggplot2* functions don't accept time series objects, so time series must be first coerced into `data frames`.

```
> library(rutils)  # Load xts time series data
> library(reshape2)
> library(ggplot2)
> pricev <- rutils::etfenv$prices[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> # Create data frame of time series
> dframe <- data.frame(datev=zoo::index(pricev), zoo::coredata(price
> # reshape data into a single column
> dframe <- reshape2::melt(dframe, id="dates")
> x11(width=6, height=5)  # Open plot window
> # ggplot the melted dframe
> ggplot(data=dframe,
+  mapping=aes(x=datev, y=value, colour=variable)) +
+  geom_line() +
+   xlab("") + ylab("") +
+   ggtitle("VTI and IEF") +
+   theme(  # Add legend and title
+     legend.position=c(0.2, 0.8),
+     plot.title=element_text(vjust=-2.0)
+   )  # end theme
```



Time series with multiple columns must be reshaped into a single column, which can be performed using the function `melt()` from package *reshape2*,

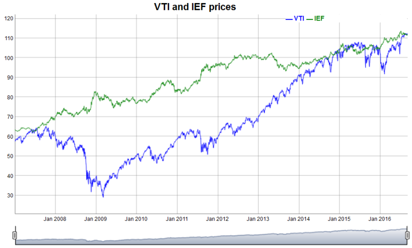
Interactive Time Series Plots Using Package *dygraphs*

The function `dygraph()` from package *dygraphs* creates interactive, zoomable plots from *xts* time series.

The function `dyOptions()` adds options (like colors, etc.) to a *dygraph* plot.

The function `dyRangeSelector()` adds a date range selector to the bottom of a *dygraphs* plot.

```
> # Load rutils which contains etfenv dataset
> library(rutils)
> library(dygraphs)
> pricev <- rutils::etfenv$prices[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> # Plot dygraph with date range selector
> dygraph(pricev, main="VTI and IEF prices") %>%
+   dyOptions(colors=c("blue","green")) %>%
+   dyRangeSelector()
```



The *dygraphs* package in R is an interface to the *dygraphs JavaScript* charting library.

Interactive *dygraphs* plots require running *JavaScript* code, which can be embedded in *html* documents, and displayed by web browsers.

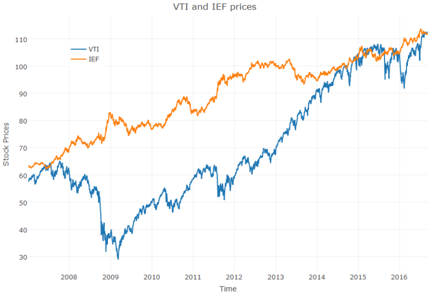
But *pdf* documents can't run *JavaScript* code, so they can't display interactive *dygraphs* plots,

Interactive Time Series Plots Using Package *plotly*

The function `plot_ly()` from package *plotly* creates interactive plots from data residing in `data frames`.

The function `add_trace()` adds elements to a *plotly* plot.

The function `layout()` modifies the layout of a *plotly* plot.




```
> # Load rutils which contains etfenv dataset
> library(rutils)
> library(plotly)
> pricev <- rutils::etfenv$prices[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> # Create data frame of time series
> dframe <- data.frame(datev=zoo::index(pricev),
+     zoo::coredata(pricev))
> # Plotly syntax using pipes
> dframe %>%
+   plot_ly(x=~datev, y=~VTI, type="scatter", mode="lines", name="VTI") %>%
+   add_trace(x=~datev, y=~IEF, type="scatter", mode="lines", name="IEF") %>%
+   layout(title="VTI and IEF prices",
+    xaxis=list(title="Time"),
+    yaxis=list(title="Stock Prices"),
+    legend=list(x=0.1, y=0.9))
> # Or use standard plotly syntax
> plotobj <- plot_ly(data=dframe, x=~datev, y=~VTI, type="scatter", mode="lines", name="VTI")
> plotobj <- add_trace(p=plotobj, x=~datev, y=~IEF, type="scatter", mode="lines", name="IEF")
> plotobj <- layout(p=plotobj, title="VTI and IEF prices", xaxis=list(title="Time"), yaxis=list(title="Stock Prices"), legend=list(x=0.
> plotobj
```

Subsetting *xts* Time Series

xts time series can be subset in similar ways as *zoo* time series.

In addition, *xts* time series can be subset using date strings, or date range strings, for example: `["2014-10-15/2015-01-10"]`.

xts time series can be subset by year, week, days, or even seconds.

If only the date is subset, then a comma `","` after the date range isn't necessary.

The function `.subset_xts()` allows fast subsetting of *xts* time series, which for large datasets can be faster than the bracket `"[]"` notation.

```
> # Subset xts using a date range string
> pricev <- rutils::etfenv$prices
> pricesub <- pricev["2014-10-15/2015-01-10", 1:4]
> first(pricesub)
> last(pricesub)
> # Subset Nov 2014 using a date string
> pricesub <- pricev["2014-11", 1:4]
> first(pricesub)
> last(pricesub)
> # Subset all data after Nov 2014
> pricesub <- pricev["2014-11/", 1:4]
> first(pricesub)
> last(pricesub)
> # Comma after date range not necessary
> all.equal(pricev["2014-11", ], pricev["2014-11"])
> # .subset_xts() is faster than the bracket []
> library(microbenchmark)
> summary(microbenchmark(
+   bracket=pricev[10:20, ],
+   subset=xts::.subset_xts(pricev, 10:20),
+   times=10))[, c(1, 4, 5)]
```

Fast Subsetting of *xts* Time Series

Subsetting of *xts* time series can be made much faster if the right operations are used.

Subsetting *xts* time series using `Boolean` vectors is usually faster than using date strings.

But the speed of subsetting can be reduced by additional operations, like coercing strings into dates.

```
> # Specify string representing a date
> datev <- "2014-10-15"
> # Subset prices in two different ways
> pricev <- rutils::etfenv$prices
> all.equal(pricev[zoo::index(pricev) >= datev],
+     pricev[paste0(datev, "/")])
> # Boolean subsetting is slower because coercing string into date
> library(microbenchmark)
> summary(microbenchmark(
+   boolean=(pricev[zoo::index(pricev) >= datev]),
+   date=(pricev[paste0(datev, "/")]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Coerce string into a date
> datev <- as.Date("2014-10-15")
> # Boolean subsetting is faster than using date string
> summary(microbenchmark(
+   boolean=(pricev[zoo::index(pricev) >= datev]),
+   date=(pricev[paste0(datev, "/")]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Subsetting Recurring *xts* Time Intervals

A *recurring time interval* is the same time interval every day, for example the time interval from 9:30AM to 4:00PM every day.

xts series can be subset on recurring time intervals using the `"T"` notation.

For example, to subset the time interval from 9:30AM to 4:00PM every day: `["T09:30:00/T16:00:00"]`

Warning messages that "timezone of object is different than current timezone" can be suppressed by calling the function options() with argument `"xts_check_tz=FALSE"`

```
> pricev <- HighFreq::SPY["2012-04"]
> # Subset recurring time interval using "T notation",
> pricev <- pricev["T10:30:00/T15:00:00"]
> first(pricev["2012-04-16"])  # First element of day
> last(pricev["2012-04-16"])  # Last element of day
> # Suppress timezone warning messages
> options(xts_check_tz=FALSE)
```

Binding *xts* Time Series by Rows

The function `rbind()` joins the rows of *xts* time series.

If the time series have overlapping time indices then the join produces duplicate rows with the same dates.

The duplicate rows can be removed using the function `duplicated()`.

The function `duplicated()` returns a `Boolean` vector indicating the duplicate elements of a vector.

The function `duplicated()` with argument `"fromLast=TRUE"` identifies duplicate elements starting from the end.

```
> # Create time series with overlapping time indices
> vti1 <- rutils::etfenv$VTI["/2015"]
> vti2 <- rutils::etfenv$VTI["2014/"]
> dates1 <- zoo::index(vti1)
> dates2 <- zoo::index(vti2)
> # Join by rows
> vti <- rbind(vti1, vti2)
> datev <- zoo::index(vti)
> sum(duplicated(datev))
> vti <- vti[!duplicated(datev), ]
> all.equal(vti, rutils::etfenv$VTI)
> # Alternative method - slightly slower
> vti <- rbind(vti1, vti2[!(zoo::index(vti2) %in% zoo::index(vti1))]
> all.equal(vti, rutils::etfenv$VTI)
> # Remove duplicates starting from the end
> vti <- rbind(vti1, vti2)
> vti <- vti[!duplicated(datev), ]
> vtifl <- vti[!duplicated(datev, fromLast=TRUE), ]
> all.equal(vti, vtifl)
```

Properties of *xts* Time Series

xts series always have a dim attribute, unlike *zoo*, which have no dim attribute when they only have one column of data.

zoo series with multiple columns have a dim attribute, and are therefore matrices.

But *zoo* with a single column don't, and are therefore vectors not matrices.

When a *zoo* is subset to a single column, the dim attribute is dropped, which can create errors.

```
> pricev <- rutils::etfenv$prices[, c("VTI", "IEF")]
> pricev <- na.omit(pricev)
> str(pricev)  # Display structure of xts
> # Subsetting zoo to single column drops dim attribute
> pricezoo <- as.zoo(pricev)
> dim(pricezoo)
> dim(pricezoo[, 1])
> # zoo with single column are vectors not matrices
> c(is.matrix(pricezoo), is.matrix(pricezoo[, 1]))
> # xts always have a dim attribute
> rbind(base=dim(pricev), subs=dim(pricev[, 1]))
> c(is.matrix(pricev), is.matrix(pricev[, 1]))
```

lag() and diff() Operations on *xts* Time Series

The methods `xts::lag()` and `xts::diff()` for *xts* series differ from those of package *zoo*.

By default, the method `xts::lag()` replaces the current value with values from the past (negative lags replace with values from the future).

The methods `zoo::lag()` and `zoo::diff()` shorten the series by the number of lag periods.

By default, the methods `xts::lag()` and `xts::diff()` retain the same number of elements, by padding with leading or trailing `NA` values.

In order to avoid padding with `NA` values, asset returns can be padded with zeros, and prices can be padded with the first or last elements of the input vector.

```
> # Lag of zoo shortens it by one row
> rbind(base=dim(pricezoo), lag=dim(lag(pricezoo)))
> # Lag of xts doesn't shorten it
> rbind(base=dim(pricev), lag=dim(lag(pricev)))
> # Lag of zoo is in opposite direction from xts
> head(lag(pricezoo, -1), 4)
> head(lag(pricev), 4)
```

Determining Calendar *End points* of *xts* Time Series

The function endpoints() from package *xts* extracts the indices of the last observations in each calendar period of time of an *xts* series.

For example:
`endpoints(x, on="hours")`
extracts the indices of the last observations in each hour.

The *end points* calculated by endpoints() aren't always equally spaced, and aren't the same as those calculated from fixed intervals.

For example, the last observations in each day aren't equally spaced due to weekends and holidays.

```
> # Indices of last observations in each hour
> endd <- xts::endpoints(pricev, on="hours")
> head(endd)
> # Extract the last observations in each hour
> head(pricev[endd, ])
```

Converting *xts* Time Series to Lower Periodicity

The function to.period() converts a time series to a lower periodicity (for example from hourly to daily periodicity).

to.period() returns a time series of open, high, low, and close values (*OHLC*) for the lower period.

to.period() converts both univariate and *OHLC* time series to a lower periodicity.

```
> # Lower the periodicity to months
> pricem <- to.period(x=pricev, period="months", name="MSFT")
> # Convert colnames to standard OHLC format
> colnames(pricem)
> colnames(pricem) <- sapply(
+   strsplit(colnames(pricem), split=".", fixed=TRUE),
+   function(namev) namev[-1]
+   )  # end sapply
> head(pricem, 3)
> # Lower the periodicity to years
> pricey <- to.period(x=pricem, period="years", name="MSFT")
> colnames(pricey) <- sapply(
+   strsplit(colnames(pricey), split=".", fixed=TRUE),
+   function(namev) namev[-1]
+   )  # end sapply
> head(pricey)
```

Plotting *OHLC* Time Series Using `chart_Series()`

The function `chart_Series()` from package *quantmod* can plot candlestick plots of *OHLC* prices.

Each *candlestick* displays one period of data, and consists of a box representing the *Open* and *Close* prices, and a vertical line representing the *High* and *Low* prices.

The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

```
> load(file="/Users/jerzy/Develop/lecture_slides/data/zoo_data.RData
> library(quantmod)  # Load package quantmod
> # as.xts() coerces zoo series into xts series
> class(zoo_stx)
> pricexts <- as.xts(zoo_stx)
> dim(pricexts)
> head(pricexts[, 1:4], 4)
> # OHLC candlechart
> plotheme <- chart_theme()
> plotheme$col$up.col <- c("green")
> plotheme$col$dn.col <- c("red")
> chart_Series(x=pricexts["2016-05/2016-06", 1:4], theme=plotheme,
+   name="Candlestick Plot of OHLC Stock Prices")
```

Plotting *OHLC* Time Series Using Package *dygraphs*

The function `dygraph()` from package *dygraphs* creates interactive plots for *xts* time series.

The function `dyCandlestick()` creates a *candlestick* plot object for *OHLC* data, and uses the first four columns to plot *candlesticks*, and it plots any additional columns as lines.

The function `dyOptions()` adds options (like colors, etc.) to a *dygraph* plot.

```
> library(dygraphs)
> # Create dygraphs object
> dyplot <- dygraphs::dygraph(pricexts["2016-05/2016-06", 1:4])
> # Convert dygraphs object to candlestick plot
> dyplot <- dygraphs::dyCandlestick(dyplot)
> # Render candlestick plot
> dyplot
> # Candlestick plot using pipes syntax
> dygraphs::dygraph(pricexts["2016-05/2016-06", 1:4]) %>%
+   dyCandlestick() %>%
+   dyOptions(colors="red", strokeWidth=3)
> # Candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dyOptions(
+   dygraphs::dygraph(pricexts["2016-05/2016-06", 1:4]),
+   colors="red", strokeWidth=3))
```

Each *candlestick* displays one period of data, and consists of a box representing the *Open* and *Close* prices, and a vertical line representing the *High* and *Low* prices.

The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

Time Series Classes in R

R and other packages contain a number of different time series classes:

- Class *ts* from base package *stats*:
  native time series class in R, but allows only *regular* (equally spaced) date-time index,
  not suitable for sophisticated financial applications,
- Class *zoo*: allows *irregular* date-time index,
  the *zoo* index can be from any *date-time* class,
- Class *xts* extension of *zoo* class: most widely accepted time series class,
  designed for high-frequency and *OHLC* data,
  contains convenient functions for plotting, calculating rolling max, min, etc.
- Class *timeSeries* from the *Rmetrics* suite,

```
> # Create zoo time series
> datev <- seq(from=as.Date("2014-07-14"), by="day", length.out=10)
> timeser <- zoo(x=sample(10), order.by=datev)
> class(timeser)
> timeser
> library(xts)
> # Coerce zoo time series to class xts
> pricexts <- as.xts(timeser)
> class(xtseries)
> xtseries
```

Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture_6.pdf*, and run all the code in *FRE6871_Lecture_6.R*

Recommended

- Read about *PCA* in:
  pca-handout.pdf
  pcaTutorial.pdf
- Read about *optimization methods*:
  Bolker Optimization Methods.pdf
  Yollin Optimization.pdf
  Boudt DEoptim Large Portfolio Optimization.pdf