FRE6871 R in Finance

Lecture#6, Spring 2024

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 29, 2024

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)
> tseries <- ts(data=datav, start=startd, frequency=tstep)
> tseries  # Display time series
> # Display index dates
> as.Date(date_decimal(zoo::coredata(time(tseries))))
> # bi-monthly geometric Brownian motion starting mid-1990
> tseries <- 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.

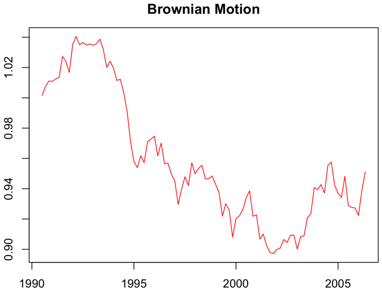
The function window() subsets the a *ts* time series object.

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

Plotting *ts* Time Series Objects

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

```
> # Create plot
> plot(tseries, 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.

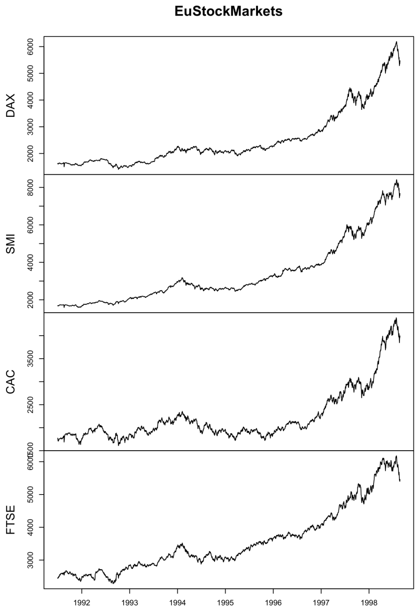
datasets is a base package containing various datasets, for example: EuStockMarkets.

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="")
```

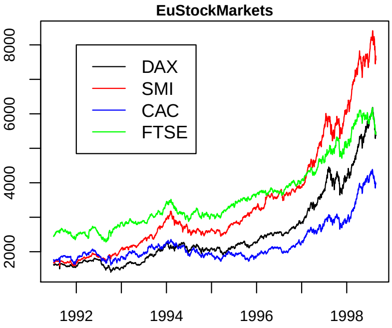


Plotting EuStockMarkets Data

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
2024-04-27 2024-04-28 2024-04-29 2024-04-30 2024-05-01 2024-05-02 20
    0.1450     0.4383     0.1532     1.0849     1.9995    -0.8119
2024-05-04 2024-05-05 2024-05-06 2024-05-07 2024-05-08
    0.5859     0.3601    -0.0253     0.1509     0.1101
> attributes(zoots)
$index
 [1] "2024-04-27" "2024-04-28" "2024-04-29" "2024-04-30" "2024-05-01
 [6] "2024-05-02" "2024-05-03" "2024-05-04" "2024-05-05" "2024-05-06
[11] "2024-05-07" "2024-05-08"

$class
[1] "zoo"
> class(zoots)  # Class "zoo"
[1] "zoo"
> tail(zoots, 3)  # Get last few elements
2024-05-06 2024-05-07 2024-05-08
   -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  0.585
[10] -0.0253  0.1509  0.1101
> zoo::index(zoots)  # Extract time index
 [1] "2024-04-27" "2024-04-28" "2024-04-29" "2024-04-30" "2024-05-0
 [6] "2024-05-02" "2024-05-03" "2024-05-04" "2024-05-05" "2024-05-0
[11] "2024-05-07" "2024-05-08"
> start(zoots)  # First date
[1] "2024-04-27"
> end(zoots)  # Last date
[1] "2024-05-08"
> zoots[start(zoots)]  # First element
2024-04-27
     0.145
> zoots[end(zoots)]  # Last element
2024-05-08
      0.11
> zoo::coredata(zoots) <- rep(1, NROW(zoots))  # Replace coredata
> cumsum(zoots)  # Cumulative sum
2024-04-27 2024-04-28 2024-04-29 2024-04-30 2024-05-01 2024-05-02 2
         1          2          3          4          5          6
2024-05-04 2024-05-05 2024-05-06 2024-05-07 2024-05-08
         8          9         10         11         12
> cummax(cumsum(zoots))
2024-04-27 2024-04-28 2024-04-29 2024-04-30 2024-05-01 2024-05-02 2
         1          2          3          4          5          6
2024-05-04 2024-05-05 2024-05-06 2024-05-07 2024-05-08
         8          9         10         11         12
> cummin(cumsum(zoots))
2024-04-27 2024-04-28 2024-04-29 2024-04-30 2024-05-01 2024-05-02 2
         1          1          1          1          1          1
2024-05-04 2024-05-05 2024-05-06 2024-05-07 2024-05-08
         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" "2013-06-1
 [6] "2013-06-20" "2013-06-21" "2013-06-22" "2013-06-23" "2013-06-2

$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.

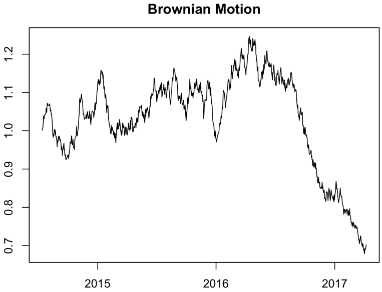
If the `"na.pad"` argument is set to `TRUE`, then they return time series of the same length, with `NA` values added where needed.

```
> 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
2013-06-17     4
```

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.

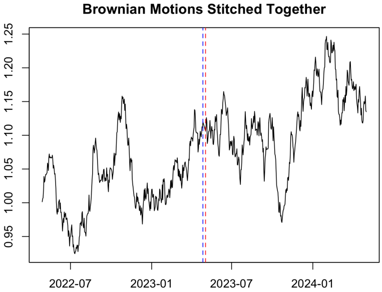
In addition, *zoo* time series can be subset using `Date` objects.

```
> # 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"
> todayv <- Sys.Date()
> index1 <- seq(todayv-2*365, by="days", length.out=365)
> # Create zoo time series of random returns
> zoots1 <- zoo(rnorm(NROW(index1)), order.by=index1)
> # Create another zoo time series of random returns
> index2 <- seq(todayv-360, by="days", length.out=365)
> zoots2 <- zoo(rnorm(NROW(index2)), order.by=index2)
> # rbind the two time series - ts1 supersedes ts2
> zootsub2 <- zoots2[zoo::index(zoots2) > end(zoots1)]
> zoots3 <- rbind(zoots1, zootsub2)
> # Plot zoo time series of geometric Brownian motion
> plot(exp(cumsum(zoots3)/100), xlab="", ylab="")
> # Add vertical lines at stitch point
> abline(v=end(zoots1), col="blue", lty="dashed")
> abline(v=start(zoots2), 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
> zoots1 <- zoo(rnorm(NROW(index1)), order.by=index1)
> # Create another zoo time series of random returns
> index2 <- Sys.Date() + -1:3
> zoots2 <- zoo(rnorm(NROW(index2)), order.by=index2)
> merge(zoots1, zoots2)  # union of dates
> # Intersection of dates
> merge(zoots1, zoots2, 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)
> rutils::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
> tseries <- as.ts(zoots)
> tsp(tseries)  # 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()
> tseries <- ts(data=datav, start=startd, frequency=tstep)
> # Display start of time series
> window(tseries, start=start(tseries),
+  end=start(tseries)+4/365)
> head(time(tseries))  # Display index dates
> head(as.Date(date_decimal(zoo::coredata(time(tseries)))))
```

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

Coercing *zoo* Time Series Into Class *xts*

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

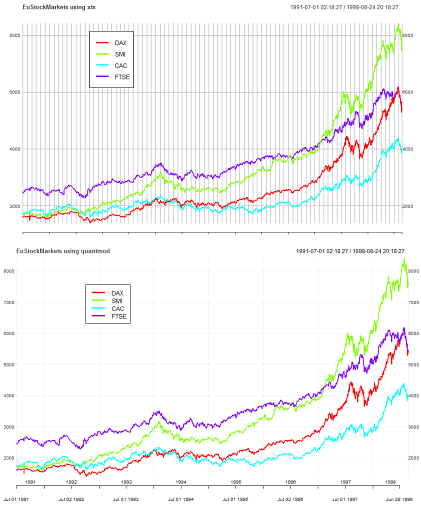
as.xts() preserves the *index* attributes of the original time series.

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")
```



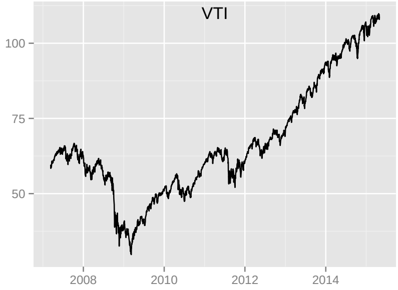
Plotting *xts* Using Package *ggplot2*

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
```

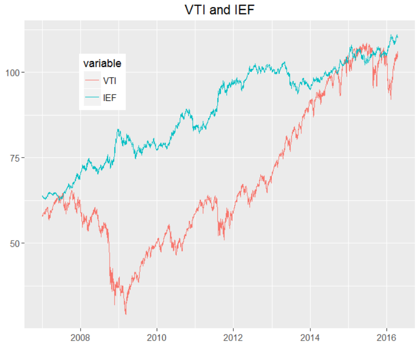


Plotting Multiple *xts* Using Package *ggplot2*

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*,

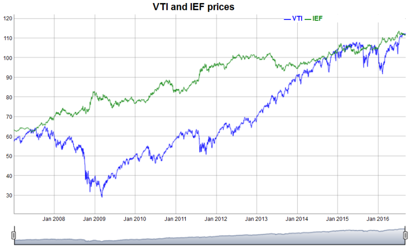
Interactive Time Series Plots Using Package *dygraphs*

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.

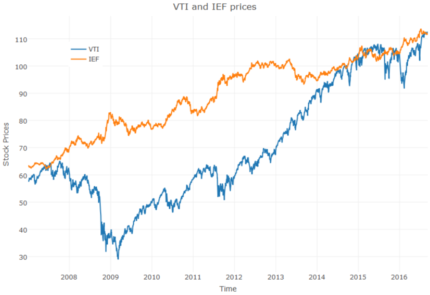
But *pdf* documents can't run *JavaScript* code, so they can't display interactive *dygraphs* plots,

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)
> tseries <- zoo(x=sample(10), order.by=datev)
> class(tseries)
> tseries
> library(xts)
> # Coerce zoo time series to class xts
> pricexts <- as.xts(tseries)
> class(xtseries)
> xtseries
```

Writing Text Strings

The function cat() concatenates strings and writes them to standard output or to files.

cat() parses its argument character string and its escape sequences ("\"), but doesn't return a value.

The function print() doesn't interpret its argument, and simply prints it to standard output and invisibly returns it.

Typing the name of an object in R implicitly calls print() on that object.

The function save() writes objects to compressed binary .RData files.

```
> cat("Enter\ttab")  # Cat() parses backslash escape sequences
> print("Enter\ttab")
>
> textv <- print("hello")
> textv  # Print() returns its argument
>
> # Create string
> textv <- "Title: My Text\nSome numbers: 1,2,3,...\nRprofile files
>
> cat(textv, file="mytext.txt")  # Write to text file
>
> cat("Title: My Text",  # Write several lines to text file
+     "Some numbers: 1,2,3,...",
+     "Rprofile files contain code executed at R startup,",
+     file="mytext.txt", sep="\n")
>
> save(textv, file="mytext.RData")  # Write to binary file
```

Displaying Numeric Data

The function `print()` displays numeric data objects, with the number of digits given by the global option `"digits"`.

The function `sprintf()` returns strings formatted from text strings and numeric data.

```
> print(pi)
[1] 3.14
> print(pi, digits=10)
[1] 3.141592654
> getOption("digits")
[1] 3
> foo <- 12
> bar <- "weeks"
> sprintf("There are %i %s in the year", foo, bar)
[1] "There are 12 weeks in the year"
```

Reading Text from Files

The function `scan()` reads text or data from a file and returns it as a vector or a list.

The function `readLines()` reads lines of text from a connection (file or console), and returns them as a vector of `character` strings.

The function `readline()` reads a single line from the console, and returns it as a `character` string.

The function `file.show()` reads text or data from a file and displays in editor.

```
> # Read text from file
> scan(file="mytext.txt", what=character(), sep="\n")
>
> # Read lines from file
> readLines(con="mytext.txt")
>
> # Read text from console
> inputv <- readline("Enter a number: ")
> class(inputv)
> # Coerce to numeric
> inputv <- as.numeric(inputv)
>
> # Read text from file and display in editor:
> # file.show("mytext.txt")
> # file.show("mytext.txt", pager="")
```

Writing and Reading *Data Frames* from *Text* Files

The functions `write.table()` and `read.table()` write and read *data frames* from text files.

`write.table()` coerces objects to *data frames* before it writes them.

`read.table()` returns a *data frame*, without coercing non-numeric values to `factors` (so no need for the option `stringsAsFactors=FALSE`).

`write.table()` and `read.table()` can be used to write and read matrices from text files, but they have to be coerced back to matrices.

`write.table()` and `read.table()` are inefficient for very large data sets.

```
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> dframe <- data.frame(type=c("rose", "daisy", "tulip"),
+   color=c("red", "white", "yellow"),
+   price=c(1.5, 0.5, 1.0),
+   row.names=c("flower1", "flower2", "flower3"))  # end data.frame
> matv <- matrix(sample(1:12), ncol=3,
+   dimnames=list(NULL, c("col1", "col2", "col3")))
> rownames(matv) <- paste("row", 1:NROW(matv), sep="")
> # Write data frame to text file, and then read it back
> write.table(dframe, file="florist.txt")
> readf <- read.table(file="florist.txt")
> readf  # A data frame
> all.equal(readf, dframe)
>
> # Write matrix to text file, and then read it back
> write.table(matv, file="matrix.txt")
> readmat <- read.table(file="matrix.txt")
> readmat  # write.table() coerced matrix to data frame
> class(readmat)
> all.equal(readmat, matv)
> # Coerce from data frame back to matrix
> readmat <- as.matrix(readmat)
> class(readmat)
> all.equal(readmat, matv)
```

Copying *Data Frames* Between the *clipboard* and R

Data frames stored in the *clipboard* can be copied into R using the function `read.table()`.

Data frames in R can be copied into the *clipboard* using the function `write.table()`.

This allows convenient copying of *data frames* between R and Excel.

Data frames can also be manipulated directly in the R spreadsheet-style data editor.

Copying and pasting between the *clipboard* and R works well on Windows, but not on MacOS. There are some workarounds for MacOS:
Copy_paste_between_R_and_clipboard

```
> # Create a data frame
> dframe <- data.frame(small=c(3, 5), medium=c(9, 11), large=c(15, 1
>
> # Launch spreadsheet-style data editor
> dframe <- edit(dframe)
>
> # Copy the data frame to clipboard
> write.table(x=dframe, file="clipboard", sep="\t")
>
> # Wrapper function for copying data frame from R into clipboard
> # by default, data is tab delimited, with a header
> write_clip <- function(data, row.names=FALSE, col.names=TRUE, ...)
+   write.table(x=data, file="clipboard", sep="\t",
+       row.names=row.names, col.names=col.names, ...)
+ }  # end write_clip
>
> write_clip(data=dframe)
>
> # Wrapper function for copying data frame from clipboard into R
> # by default, data is tab delimited, with a header
> read_clip <- function(file="clipboard", sep="\t", header=TRUE, ..
+   read.table(file=file, sep=sep, header=header, ...)
+ }  # end read_clip
>
> dframe <- read.table("clipboard", header=TRUE)
> dframe <- read_clip()
```

Writing and Reading *Data Frames* From .csv Files

The easiest way to share data between R and Excel is through .csv files.

The functions write.csv() and read.csv() write and read *data frames* from .csv format files.

The functions write.csv() and read.csv() write and read *data frames* from .csv format files.

These functions are *wrappers* for write.table() and read.table().

read.csv() doesn't coerce non-numeric values to factors, so no need for the option stringsAsFactors=FALSE.

read.csv() reads row names as an extra column, unless the row.names=1 argument is used.

The argument "row.names" accepts either the number or the name of the column containing the row names.

The *.csv() functions are very inefficient for large data sets.

```
> # Write data frame to CSV file, and then read it back
> write.csv(dframe, file="florist.csv")
> readf <- read.csv(file="florist.csv")
> readf  # the row names are read in as extra column
> # Restore row names
> rownames(readf) <- readf[, 1]
> readf <- readf[, -1]  # Remove extra column
> readf
> all.equal(readf, dframe)
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> readf
> all.equal(readf, dframe)
```

Writing and Reading *Data Frames* From .csv Files (cont.)

The functions write.csv() and read.csv() can write and read *data frames* from .csv format files *without using row names*.

Row names can be omitted from the output file by calling write.csv() with the argument row.names=FALSE.

```
> # Write data frame to CSV file, without row names
> write.csv(dframe, row.names=FALSE, file="florist.csv")
> readf <- read.csv(file="florist.csv")
> readf  # A data frame without row names
> all.equal(readf, dframe)
```

Reading Data From Very Large .csv Files

Data from very large .csv files can be read in small chunks instead of all at once.

The function file() opens a connection to a file or an internet website URL.

The function read.csv() with the argument "nrows" reads only the specified number of rows from a connection and returns a *data frame*. The connection pointer is reset to the next row.

The function read.csv() with the argument "nrows" allows reading data sequentially from very large files that wouldn't fit into memory.

```
> # Open a read connection to a file
> con_read = file("/Users/jerzy/Develop/lecture_slides/data/etf_pric
> # Read the first 10 rows
> data10 <- read.csv(con_read, nrows=10)
> # Read another 10 rows
> data20 <- read.csv(con_read, nrows=10, header=FALSE)
> colnames(data20) <- colnames(data10)
> # Close the connection to the file
> close(con_read)
> # Open a read connection to a file
> con_read = file("/Users/jerzy/Develop/lecture_slides/data/etf_pric
> # Read the first 1000 rows
> data10 <- read.csv(con_read, nrows=1e3)
> colnamev <- colnames(data10)
> # Write to a file
> countv <- 1
> write.csv(data10, paste0("/Users/jerzy/Develop/data/temp/etf_price
> # Read remaining rows in a loop 10 rows at a time
> # Can produce error without getting to end of file
> while (isOpen(con_read)) {
+   datav <- read.csv(con_read, nrows=1e3)
+   colnames(datav) <- colnamev
+   write.csv(datav, paste0("/Users/jerzy/Develop/data/temp/etf_pric
+   countv <- countv + 1
+ }  # end while
```

Writing and Reading Matrices From .csv Files

The functions `write.csv()` and `read.csv()` can write and read matrices from `.csv` format files.

If row names can be omitted in the output file, then `write.csv()` can be called with argument `row.names=FALSE`.

If the input file doesn't contain row names, then `read.csv()` can be called without the `"row.names"` argument.

```
> # Write matrix to csv file, and then read it back
> write.csv(matv, file="matrix.csv")
> readmat <- read.csv(file="matrix.csv", row.names=1)
> readmat  # Read.csv() reads matrix as data frame
> class(readmat)
> readmat <- as.matrix(readmat)  # Coerce to matrix
> all.equal(readmat, matv)
> write.csv(matv, row.names=FALSE,
+     file="matrix_ex_rows.csv")
> readmat <- read.csv(file="matrix_ex_rows.csv")
> readmat <- as.matrix(readmat)
> readmat  # A matrix without row names
> all.equal(readmat, matv)
```

Writing and Reading Matrices (cont.)

There are several ways of writing and reading matrices from `.csv` files, with tradeoffs between simplicity, data size, and speed.

The function `write.matrix()` writes a matrix to a text file, without its row names.

`write.matrix()` is part of package *MASS*.

The advantage of function `scan()` is its speed, but it doesn't handle row names easily.

Removing row names simplifies the writing and reading of matrices.

The function `readLines` reads whole lines and returns them as single strings.

```
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> library(MASS)  # Load package "MASS"
> # Write to CSV file by row - it's very SLOW!!!
> MASS::write.matrix(matv, file="matrix.csv", sep=",")
> # Read using scan() and skip first line with colnames
> readmat <- scan(file="matrix.csv", sep=",", skip=1,
+   what=numeric())
> # Read colnames
> colnamev <- readLines(con="matrix.csv", n=1)
> colnamev  # this is a string!
> # Convert to char vector
> colnamev <- strsplit(colnamev, split=",")[[1]]
> readmat  # readmat is a vector, not matrix!
> # Coerce by row to matrix
> readmat <- matrix(readmat, ncol=NROW(colnamev), byrow=TRUE)
> # Restore colnames
> colnames(readmat) <- colnamev
> readmat
> # Scan() is a little faster than read.csv()
> library(microbenchmark)
> summary(microbenchmark(
+   read_csv=read.csv("matrix.csv"),
+   scan=scan(file="matrix.csv", sep=",",
+     skip=1, what=numeric()),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Reading Matrices Containing Bad Data

Very often data that is read from external sources contains elements with bad data.

An example of bad data are character strings within sets of numeric data.

Columns of numeric data that contain strings are coerced to character or factor, when they're read by read.csv().

The function as.numeric() coerces complex data objects into numeric vectors, and removes all their *attributes*.

as.numeric() coerces strings that don't represent numbers into NA values.

```
> # Read data from a csv file, including row names
> matv <- read.csv(file="matrix_bad.csv", row.names=1)
> matv
> class(matv)
> # Columns with bad data are character or factor
> sapply(matv, class)
> # Coerce character column to numeric
> matv$col2 <- as.numeric(matv$col2)
> # Or
> # Copy row names
> rownames <- row.names(matv)
> # sapply loop over columns and coerce to numeric
> matv <- sapply(matv, as.numeric)
> # Restore row names
> row.names(matv) <- rownames
> # Replace NAs with zero
> matv[is.na(matv)] <- 0
> # matrix without NAs
> matv
```

Writing and Reading Time Series From *Text* Files

The package *zoo* contains functions `write.zoo()` and `read.zoo()` for writing and reading *zoo* time series from `.txt` and `.csv` files.

The functions `write.zoo()` and `read.zoo()` are *wrappers* for `write.table()` and `read.table()`.

The function `write.zoo()` writes the *zoo* series index as a character string in quotations `""`, to make it easier to read (parse) by `read.zoo()`.

Users may also directly use `write.table()` and `read.table()`, instead of `write.zoo()` and `read.zoo()`.

```
> library(zoo)  # Load package zoo
> # Create zoo with Date index
> datev <- seq(from=as.Date("2013-06-15"), by="day",
+        length.out=100)
> pricev <- zoo(rnorm(NROW(datev)), order.by=datev)
> head(pricev, 3)
> # Write zoo series to text file, and then read it back
> write.zoo(pricev, file="pricev.txt")
> pricezoo <- read.zoo("pricev.txt")  # Read it back
> all.equal(pricezoo, pricev)
> # Perform the same using write.table() and read.table()
> # First coerce pricev into data frame
> dframe <- as.data.frame(pricev)
> dframe <- cbind(datev, dframe)
> # Write pricev to text file using write.table
> write.table(dframe, file="pricev.txt",
+       row.names=FALSE, col.names=FALSE)
> # Read data frame from file
> pricezoo <- read.table(file="pricev.txt")
> sapply(pricezoo, class)  # A data frame
> # Coerce data frame into pricev
> pricezoo <- zoo::zoo(
+   drop(as.matrix(pricezoo[, -1])),
+   order.by=as.Date(pricezoo[, 1]))
> all.equal(pricezoo, pricev)
```

Writing and Reading Time Series From .csv Files

By default the functions `zoo::write.zoo()` and `zoo::read.zoo()` write data in *space*-delimited text format, but they can also write to *comma*-delimited `.csv` files by passing the parameter `sep=","`.

Single column *zoo* time series usually don't have a dimension attribute, and they don't have a column name, unlike multi-column *zoo* time series, and this can cause hard to detect bugs.

It's best to always pass the argument `"col.names=TRUE"` to the function `write.zoo()`, to make sure it writes a column name for a single column *zoo* time series.

Reading a `.csv` file containing a single column of data using the function `read.zoo()` produces a *zoo* time series with a `NULL` dimension, unless the argument `"drop=FALSE"` is passed to `read.zoo()`.

Users may also directly use `write.table()` and `read.table()`, instead of `write.zoo()` and `read.zoo()`.

```
> # Write zoo series to CSV file, and then read it back
> write.zoo(pricev, file="pricev.csv", sep=",", col.names=TRUE)
> pricezoo <- read.zoo(file="pricev.csv",
+   header=TRUE, sep=",", drop=FALSE)
> all.equal(pricev, drop(pricezoo))
```

Writing and Reading Time Series With *Date-time* Index

The function `read.csv.zoo()` reads *zoo* time series from `.csv` files.

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

If the index of a *zoo* time series is a *date-time*, then `write.zoo()` writes the date and time fields as character strings separated by a *space* between them, inside quotations `""`.

Very often `.csv` files contain custom *date-time* formats, which need to be passed as parameters into `read.zoo()` for proper formatting.

The `"FUN"` argument of `read.zoo()` accepts a function for coercing the date and time columns of the input data into a *date-time* object suitable for the *zoo* index.

The function `as.POSIXct()` coerces character strings into `POSIXct` *date-time* objects.

```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Create zoo with POSIXct date-time index
> datev <- seq(from=as.POSIXct("2013-06-15"),
+         by="hour", length.out=100)
> pricev <- zoo(rnorm(NROW(datev)), order.by=datev)
> head(pricev, 3)
> # Write zoo series to CSV file, and then read it back
> write.zoo(pricev, file="pricev.csv", sep=",", col.names=TRUE)
> # Read from CSV file using read.csv.zoo()
> pricezoo <- read.csv.zoo(file="pricev.csv")
> all.equal(pricev, pricezoo)
> # Coerce to xts series
> xtsv <- xts::as.xts(pricezoo)
> class(xtsv); head(xtsv, 3)
> # Coerce zoo series into data frame with custom date format
> dframe <- as.data.frame(pricev)
> dframe <- cbind(format(datev, "%m-%d-%Y %H:%M:%S"), dframe)
> head(dframe, 3)
> # Write zoo series to csv file using write.table
> write.table(dframe, file="pricev.csv",
+       sep=",", row.names=FALSE, col.names=FALSE)
> # Read from CSV file using read.csv.zoo()
> pricezoo <- read.zoo(file="pricev.csv",
+   header=FALSE, sep=",", FUN=as.POSIXct,
+   format="%m-%d-%Y %H:%M:%S", tz="America/New_York")
> # Or using read.csv.zoo()
> pricezoo <- read.csv.zoo(file="pricev.csv", header=FALSE,
+   format="%m-%d-%Y %H:%M:%S", tz="America/New_York")
> head(pricezoo, 3)
> all.equal(pricev, pricezoo)
```

Reading Time Series With Numeric *Date-time* Index

If the index of a time series is `numeric` (representing the *moment of time*, either as the number of days or seconds), then it must be coerced to a proper *date-time* class.

A convenient way of reading time series with a numeric index is by using `read.table()`, and then coercing the *data frame* into a time series.

The function `as.POSIXct.numeric()` coerces a `numeric` value representing the *moment of time* into a `POSIXct` *date-time*, equal to the *clock time* in the local *time zone*.

```
> # Read time series from CSV file, with numeric date-time
> datazoo <- read.table(file="/Users/jerzy/Develop/lecture_slides/da
+   header=TRUE, sep=",")
> # A data frame
> class(datazoo)
> sapply(datazoo, class)
> # Coerce data frame into xts series
> datazoo <- xts::xts(as.matrix(datazoo[, -1]),
+   order.by=as.POSIXct.numeric(datazoo[, 1], tz="America/New_York",
+                              origin="1970-01-01"))
> # An xts series
> class(datazoo)
> head(datazoo, 3)
```

Passing Arguments to the save() Function

The function `save()` writes objects to a binary file.

Object names can be passed into `save()` either through the `"..."` argument, or the `"list"` argument.

Objects passed through the `"..."` argument are not evaluated, so they must be either object names or character strings.

Object names aren't surrounded by quotes `""`, while character strings that represent object names are surrounded by quotes `""`.

Objects passed through the `"list"` argument are evaluated, so they may be variables containing character strings.

```
> var1 <- 1; var2 <- 2
> ls()  # List all objects
[1] "var1" "var2"
> ls()[1]  # List first object
[1] "var1"
> args(save)  # List arguments of save function
function (..., list = character(), file = stop("'file' must be speci
    ascii = FALSE, version = NULL, envir = parent.frame(), compress
    compression_level, eval.promises = TRUE, precheck = TRUE)
NULL
> # Save "var1" to a binary file using string argument
> save("var1", file="my_data.RData")
> # Save "var1" to a binary file using object name
> save(var1, file="my_data.RData")
> # Save multiple objects
> save(var1, var2, file="my_data.RData")
> # Save first object in list by passing to "..." argument
> # ls()[1] is not evaluated
> save(ls()[1], file="my_data.RData")

Error in save(ls()[1], file = "my_data.RData"):  object 'ls()[1]'
not found

> # Save first object in list by passing to "list" argument
> save(list=ls()[1], file="my_data.RData")
> # Save whole list by passing it to the "list" argument
> save(list=ls(), file="my_data.RData")
```

Writing and Reading Lists of Objects

The function `load()` reads data from `.RData` files, and *invisibly* returns a vector of names of objects created in the workspace.

The vector of names can be used to manipulate the objects in loops, or to pass them to functions.

```
> rm(list=ls())  # Remove all objects
> # Load objects from file
> loadobj <- load(file="my_data.RData")
> loadobj  # vector of loaded objects
> ls()  # List objects
> # Assign new values to objects in  global environment
> sapply(loadobj, function(symboln) {
+   assign(symboln, runif(1), envir=globalenv())
+ })  # end sapply
> ls()  # List objects
> # Assign new values to objects using for loop
> for (symboln in loadobj) {
+   assign(symboln, runif(1))
+ }  # end for
> ls()  # List objects
> # Save vector of objects
> save(list=loadobj, file="my_data.RData")
> # Remove only loaded objects
> rm(list=loadobj)
> # Remove the object "loadobj"
> rm(loadobj)
```

Saving Output of R to a File

The function `sink()` diverts R *text* output (excluding graphics) to a file, or ends the diversion.

Remember to call `sink()` to end the diversion!

The function `pdf()` diverts graphics output to a *pdf* file (text output isn't diverted), in vector graphics format.

The functions `png()`, `jpeg()`, `bmp()`, and `tiff()` divert graphics output to graphics files (text output isn't diverted).

The function `dev.off()` ends the diversion.

```
> sink("sinkdata.txt")# Redirect text output to file
>
> cat("Redirect text output from R\n")
> print(runif(10))
> cat("\nEnd data\nbye\n")
>
> sink()  # turn redirect off
>
> pdf("Rgraph.pdf", width=7, height=4)  # Redirect graphics to pdf f
>
> cat("Redirect data from R into pdf file\n")
> myvar <- seq(-2*pi, 2*pi, len=100)
> plot(x=myvar, y=sin(myvar), main="Sine wave",
+    xlab="", ylab="", type="l", lwd=2, col="red")
> cat("\nEnd data\nbye\n")
>
> dev.off()  # turn pdf output off
>
> png("r_plot.png")  # Redirect graphics output to png file
>
> cat("Redirect graphics from R into png file\n")
> plot(x=myvar, y=sin(myvar), main="Sine wave",
+  xlab="", ylab="", type="l", lwd=2, col="red")
> cat("\nEnd data\nbye\n")
>
> dev.off()  # turn png output off
```

Downloading *ts* Time Series Using *tseries*

`get.hist.quote()` can download daily historical data in *ts* format using the argument `"retclass="ts"`.

`get.hist.quote()` returns a *ts* object with a `frequency=1`, implying a *"day"* time unit, instead of a *"year"* time unit suitable for *year-fraction* dates.

The *ts* contains `NA` values for weekends and holidays.

```
> library(tseries)  # Load package tseries
> # Download MSFT data in ts format
> pricemsft <- suppressWarnings(
+   get.hist.quote(
+     instrument="MSFT",
+     start=Sys.Date()-3*365,
+     end=Sys.Date(),
+     retclass="ts",
+     quote=c("Open","High","Low","Close",
+       "AdjClose","Volume"),
+     origin="1970-01-01")
+ )  # end suppressWarnings
> # Calculate price adjustment vector
> ratio <- as.numeric(pricemsft[, "AdjClose"]/pricemsft[, "Close"])
> # Adjust OHLC prices
> pricemsftadj <- pricemsft
> pricemsftadj[, c("Open","High","Low","Close")] <-
+   ratio*pricemsft[, c("Open","High","Low","Close")]
> # Inspect the data
> tsp(pricemsftadj)  # frequency=1
> head(time(pricemsftadj))
> head(pricemsftadj)
> tail(pricemsftadj)
```

Downloading *zoo* Time Series Using *tseries*

The function get.hist.quote() downloads historical data from online sources.

The "provider" argument determines the *online source*, and its default value is c("yahoo", "oanda").

The "retclass" argument determines the *return class*, and its default value is c("zoo", "its", "ts").

The "quote" argument determines the data fields, and its default value is c("Open", "High", "Low", "Close").

The "AdjClose" data field is for the *Close* price adjusted for stock splits and dividends.

```
> # Download MSFT data
> pricezoo <- suppressWarnings(
+   get.hist.quote(
+     instrument="MSFT",
+     start=Sys.Date()-3*365,
+     end=Sys.Date(),
+     quote=c("Open","High","Low","Close",
+       "AdjClose","Volume"),
+     origin="1970-01-01")
+ )  # end suppressWarnings
> class(pricezoo)
> dim(pricezoo)
> head(pricezoo, 4)
```

Adjusting *OHLC* Data

Stock prices experience jumps due to stock splits and dividends.

Adjusted stock prices are stock prices that have been adjusted so they don't have jumps.

OHLC data can be adjusted for stock splits and dividends.

```
> # Calculate price adjustment vector
> ratio <- as.numeric(pricezoo[, "AdjClose"]/pricezoo[, "Close"])
> head(ratio, 5)
> tail(ratio, 5)
> # Adjust OHLC prices
> pricedj <- pricezoo
> pricedj[, c("Open","High","Low","Close")] <-
+   ratio*pricezoo[, c("Open","High","Low","Close")]
> head(pricedj)
> tail(pricedj)
```

Downloading Data From *Oanda* Using *tseries*

Oanda is a foreign exchange broker that also provides free historical currency rates data.

The function `get.hist.quote()` downloads historical data from online sources.

The `"provider"` argument determines the *online source*, and its default value is `c("yahoo", "oanda")`.

The `"retclass"` argument determines the *return class*, and its default value is `c("zoo", "its", "ts")`.

The `"quote"` argument determines the data fields, and its default value is `c("Open", "High", "Low", "Close")`.

The function `complete.cases()` returns `TRUE` if a row has no `NA` values.

```
> # Download EUR/USD data
> priceur <- suppressWarnings(
+   get.hist.quote(
+     instrument="EUR/USD",
+     provider="oanda",
+     start=Sys.Date()-3*365,
+     end=Sys.Date(),
+     origin="1970-01-01")
+ )  # end suppressWarnings
> # Bind and scrub data
> pricecombo <- cbind(priceur, pricezoo[, "AdjClose"])
> colnames(pricecombo) <- c("EURUSD", "MSFT")
> pricecombo <- pricecombo[complete.cases(pricecombo),]
> save(pricezoo, pricedj,
+      pricemsft, pricemsftadj,
+      priceur, pricecombo,
+      file="/Users/jerzy/Develop/lecture_slides/data/zoo_data.RData
> # Inspect the data
> class(priceur)
> head(priceur, 4)
```

Downloading Stock Prices Using *tseries*

Data for multiple symbols can be downloaded in an `lapply()` loop, which calls the function `tseries::get.hist.quote`.

If the body of an `apply()` loop returns a *zoo* or *xts* series, then the loop will produce an error, because `apply()` attempts to coerce its output into a vector or matrix.

So `lapply()` should be used instead of `apply()`.

The functional `lapply()` applies a function to a list of objects and returns a list of objects.

The list of *zoo* time series can be flattened into a single *zoo* series using functions `do.call()` and `cbind()`.

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

The function `do_call()` from package *rutils* performs the same operation as `do.call()`, but using recursion, which is much faster and uses less memory.

```
> # Download price and volume data for symbolv into list of zoo obje
> pricev <- suppressWarnings(
+   lapply(symbolv, # Loop for loading data
+     get.hist.quote,
+     quote=c("AdjClose", "Volume"),
+     start=Sys.Date()-3650,
+     end=Sys.Date(),
+     origin="1970-01-01")  # end lapply
+ )  # end suppressWarnings
> # Flatten list of zoo objects into a single zoo object
> pricev <- rutils::do_call(cbind, pricev)
> # Or
> # pricev <- do.call(cbind, pricev)
> # Assign names in format "symboln.Close", "symboln.Volume"
> names(pricev) <- as.numeric(sapply(symbolv,
+     paste, c("Close", "Volume"), sep="."))
> # Save pricev to a comma-separated CSV file
> write.zoo(pricev, file="pricev.csv", sep=",")
> # Save pricev to a binary .RData file
> save(pricev, file="pricev.RData")
```

The *ETF* Database

Exchange-traded Funds (*ETFs*) are funds which invest in portfolios of assets, such as stocks, commodities, or bonds.

ETFs are shares in portfolios of assets, and they are traded just like stocks.

ETFs provide investors with convenient, low cost, and liquid instruments to invest in various portfolios of assets.

The file `etf_list.csv` contains a database of exchange-traded funds (*ETFs*) and exchange traded notes (*ETNs*).

We will select a portfolio of *ETFs* for illustrating various investment strategies.

```
> # Select ETF symbols for asset allocation
> symbolv <- c("VTI", "VEU", "EEM", "XLY", "XLP", "XLE", "XLF",
+  "XLV", "XLI", "XLB", "XLK", "XLU", "VYM", "IVW", "IWB", "IWD",
+  "IWF", "IEF", "TLT", "VNQ", "DBC", "GLD", "USO", "VXX", "SVXY",
+  "MTUM", "IVE", "VLUE", "QUAL", "VTV", "USMV", "AIEQ", "QQQ")
> # Read etf database into data frame
> etflist <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data
> rownames(etflist) <- etflist$Symbol
> # Select from etflist only those ETF's in symbolv
> etflist <- etflist[symbolv, ]
> # Shorten names
> etfnames <- sapply(etflist$Name, function(name) {
+   namesplit <- strsplit(name, split=" ")[[1]]
+   namesplit <- namesplit[c(-1, -NROW(namesplit))]
+   name_match <- match("Select", namesplit)
+   if (!is.na(name_match))
+     namesplit <- namesplit[-name_match]
+   paste(namesplit, collapse=" ")
+ })  # end sapply
> etflist$Name <- etfnames
> etflist["IEF", "Name"] <- "10 year Treasury Bond Fund"
> etflist["TLT", "Name"] <- "20 plus year Treasury Bond Fund"
> etflist["XLY", "Name"] <- "Consumer Discr. Sector Fund"
> etflist["EEM", "Name"] <- "Emerging Market Stock Fund"
> etflist["MTUM", "Name"] <- "Momentum Factor Fund"
> etflist["SVXY", "Name"] <- "Short VIX Futures"
> etflist["VXX", "Name"] <- "Long VIX Futures"
> etflist["DBC", "Name"] <- "Commodity Futures Fund"
> etflist["USO", "Name"] <- "WTI Oil Futures Fund"
> etflist["GLD", "Name"] <- "Physical Gold Fund"
```

ETF Database for Investment Strategies

The database contains *ETFs* representing different *industry sectors* and *investment styles*.

The *ETFs* with names *X** represent industry *sector funds* (energy, financial, etc.)

The *ETFs* with names *I** represent *style funds* (value, growth, size).

IWB is the Russell 1000 small-cap fund.

The *SPY ETF* owns the *S&P500* index constituents. *SPY* is the biggest, the most liquid, and the oldest ETF. SPY has over $400 billion of shares outstanding, and trades over $20 billion per day, at a bid-ask spread of only one tick (cent=`$0.01`, or about `0.0022`%).

The *QQQ ETF* owns the *Nasdaq-100* index constituents.

MTUM is an *ETF* which owns a stock portfolio representing the *momentum factor*.

DBC is an *ETF* providing the total return on a portfolio of commodity futures.

Symbol	Name	Fund.Type
VTI	Total Stock Market	US Equity ETF
VEU	FTSE All World Ex US	Global Equity ETF
EEM	Emerging Market Stock Fund	Global Equity ETF
XLY	Consumer Discr. Sector Fund	US Equity ETF
XLP	Consumer Staples Sector Fund	US Equity ETF
XLE	Energy Sector Fund	US Equity ETF
XLF	Financial Sector Fund	US Equity ETF
XLV	Health Care Sector Fund	US Equity ETF
XLI	Industrial Sector Fund	US Equity ETF
XLB	Materials Sector Fund	US Equity ETF
XLK	Technology Sector Fund	US Equity ETF
XLU	Utilities Sector Fund	US Equity ETF
VYM	Large-cap Value	US Equity ETF
IVW	S&P 500 Growth Index Fund	US Equity ETF
IWB	Russell 1000	US Equity ETF
IWD	Russell 1000 Value	US Equity ETF
IWF	Russell 1000 Growth	US Equity ETF
IEF	10 year Treasury Bond Fund	US Fixed Income ETF
TLT	20 plus year Treasury Bond Fund	US Fixed Income ETF
VNQ	REIT ETF - DNQ	US Equity ETF
DBC	Commodity Futures Fund	Commodity Based ETF
GLD	Physical Gold Fund	Commodity Based ETF
USO	WTI Oil Futures Fund	Commodity Based ETF
VXX	Long VIX Futures	Commodity Based ETN
SVXY	Short VIX Futures	Commodity Based ETF
MTUM	Momentum Factor Fund	US Equity ETF
IVE	S&P 500 Value Index Fund	US Equity ETF
VLUE	MSCI USA Value Factor	US Equity ETF
QUAL	MSCI USA Quality Factor	US Equity ETF
VTV	Value	US Equity ETF
USMV	MSCI USA Minimum Volatility Fund	US Equity ETF
AIEQ	AI Powered Equity	US Asset Allocation ET
QQQ	QQQ Trust	US Equity ETF

Exchange Traded Notes (*ETNs*)

ETNs are similar to *ETFs*, with the difference that *ETFs* are shares in a fund which owns the underlying assets, while *ETNs* are notes from issuers which promise payouts according to a formula tied to the underlying asset.

ETFs are similar to mutual funds, while *ETNs* are similar to corporate bonds.

ETNs are technically unsecured corporate debt, but instead of fixed coupons, they promise to provide returns on a market index or futures contract.

The *ETN* issuer promises the payout and is responsible for tracking the index.

The *ETN* investor has counterparty credit risk to the *ETN* issuer.

VXX is an *ETN* providing the total return of *long VIX* futures contracts (specifically the *S&P* VIX Short-Term Futures Index).

VXX is *bearish* because it's *long* VIX futures, and the VIX *rises* when stock prices *drop*.

SVXY is an *ETF* providing the total return of *short VIX* futures contracts.

SVXY is *bullish* because it's *short* VIX futures, and the VIX *drops* when stock prices *rise*.

Downloading ETF Prices Using Package *quantmod*

The function `getSymbols()` downloads time series data into the specified *environment*.

`getSymbols()` downloads the daily *OHLC* prices and trading volume (Open, High, Low, Close, Adjusted, Volume).

`getSymbols()` creates objects in the specified *environment* from the input strings (names), and assigns the data to those objects, without returning them as a function value, as a *side effect*.

If the argument `"auto.assign"` is set to `FALSE`, then `getSymbols()` returns the data, instead of assigning it silently.

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Tiingo* and *Alpha Vantage* as the only major providers of free daily *OHLC* stock prices.

But Quandl doesn't provide free *ETF* prices, leaving *Alpha Vantage* as the best provider of free daily *ETF* prices.

```
> # Select ETF symbols for asset allocation
> symbolv <- c("VTI", "VEU", "EEM", "XLY", "XLP", "XLE", "XLF",
+  "XLV", "XLI", "XLB", "XLK", "XLU", "VYM", "IVW", "IWB", "IWD",
+  "IWF", "IEF", "TLT", "VNQ", "DBC", "GLD", "USO", "VXX", "SVXY",
+  "MTUM", "IVE", "VLUE", "QUAL", "VTV", "USMV", "AIEQ", "QQQ")
> library(rutils)  # Load package rutils
> etfenv <- new.env()  # New environment for data
> # Boolean vector of symbols already downloaded
> isdownloaded <- symbolv %in% ls(etfenv)
> # Download data for symbolv using single command - creates pacing
> getSymbols.av(symbolv, adjust=TRUE, env=etfenv,
+   output.size="full", api.key="T7JPW54ES8G75310")
> # Download data from Alpha Vantage using while loop
> nattempts <- 0  # number of download attempts
> while ((sum(!isdownloaded) > 0) & (nattempts<10)) {
+   # Download data and copy it into environment
+   nattempts <- nattempts + 1
+   cat("Download attempt = ", nattempts, "\n")
+   for (symboln in na.omit(symbolv[!isdownloaded][1:5])) {
+     cat("Processing: ", symboln, "\n")
+     tryCatch(  # With error handler
+ quantmod::getSymbols.av(symboln, adjust=TRUE, env=etfenv, auto.ass
+ # Error handler captures error condition
+ error=function(msg) {
+   print(paste0("Error handler: ", msg))
+ },  # end error handler
+ finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdownloaded <- symbolv %in% ls(etfenv)
+   cat("Pausing 1 minute to avoid pacing...\n")
+   Sys.sleep(65)
+ }  # end while
> # Download all symbolv using single command - creates pacing erro
> # quantmod::getSymbols.av(symbolv, env=etfenv, adjust=TRUE, from="
```

Inspecting ETF Prices in an Environment

The function `get()` retrieves objects that are referenced using character strings, instead of their names.

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list.

```
> ls(etfenv)  # List files in etfenv
> # Get class of object in etfenv
> class(get(x=symbolv[1], envir=etfenv))
> # Another way
> class(etfenv$VTI)
> colnames(etfenv$VTI)
> # Get first 3 rows of data
> head(etfenv$VTI, 3)
> # Get last 11 rows of data
> tail(etfenv$VTI, 11)
> # Get class of all objects in etfenv
> eapply(etfenv, class)
> # Get class of all objects in R workspace
> lapply(ls(), function(namev) class(get(namev)))
> # Get end dates of all objects in etfenv
> as.Date(sapply(etfenv, end))
```

Adjusting Stock Prices Using Package *quantmod*

Traded stock and bond prices experience jumps after splits and dividends, and must be adjusted to account for them.

The function `adjustOHLC()` adjusts *OHLC* prices.

The function `get()` retrieves objects that are referenced using character strings, instead of their names.

The function `assign()` assigns a value to an object in a specified *environment*, by referencing it using a character string (name).

The functions `get()` and `assign()` allow retrieving and assigning values to objects that are referenced using character strings.

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects extracted from an *environment*.

If the argument `"adjust"` in function `getSymbols()` is set to `TRUE`, then `getSymbols()` returns adjusted data.

```
> # Check of object is an OHLC time series
> is.OHLC(etfenv$VTI)
> # Adjust single OHLC object using its name
> etfenv$VTI <- adjustOHLC(etfenv$VTI, use.Adjusted=TRUE)
>
> # Adjust OHLC object using string as name
> assign(symbolv[1], adjustOHLC(
+     get(x=symbolv[1], envir=etfenv), use.Adjusted=TRUE),
+   envir=etfenv)
>
> # Adjust objects in environment using vector of strings
> for (symboln in ls(etfenv)) {
+   assign(symboln,
+    adjustOHLC(get(symboln, envir=etfenv), use.Adjusted=TRUE),
+    envir=etfenv)
+ }  # end for
```

Extracting Time Series from Environments

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects extracted from an *environment*.

The extractor (accessor) functions from package *quantmod*: `Cl()`, `Vo()`, etc., extract columns from *OHLC* data.

A list of *xts* series can be flattened into a single *xts* series using the function `do.call()`.

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.

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list.

Time series can also be extracted from an *environment* by coercing it into a `list`, and then subsetting and merging it into an *xts* series using the function `do.call()`.

```
> library(rutils)  # Load package rutils
> # Define ETF symbols
> symbolv <- c("VTI", "VEU", "IEF", "VNQ")
> # Extract symbolv from rutils::etfenv
> pricev <- mget(symbolv, envir=rutils::etfenv)
> # pricev is a list of xts series
> class(pricev)
> class(pricev[[1]])
> tail(pricev[[1]])
> # Extract close prices
> pricev <- lapply(pricev, quantmod::Cl)
> # Collapse list into time series the hard way
> prices2 <- cbind(pricev[[1]], pricev[[2]], pricev[[3]], pricev[[4
> class(pricev2)
> dim(pricev2)
> # Collapse list into time series using do.call()
> pricev <- do.call(cbind, pricev)
> all.equal(pricev2, pricev)
> class(pricev)
> dim(pricev)
> # Or extract and cbind in single step
> pricev <- do.call(cbind, lapply(
+   mget(symbolv, envir=rutils::etfenv), quantmod::Cl))
> # Or extract and bind all data, subset by symbolv
> pricev <- lapply(symbolv, function(symboln) {
+     quantmod::Cl(get(symboln, envir=rutils::etfenv))
+ })  # end lapply
> # Or loop over etfenv without anonymous function
> pricev <- do.call(cbind,
+   lapply(as.list(rutils::etfenv)[symbolv], quantmod::Cl))
> # Same, but works only for OHLC series - produces error
> pricev <- do.call(cbind,
+   eapply(rutils::etfenv, quantmod::Cl)[symbolv])
```

Managing Time Series

Time series columns can be renamed, and then saved into `.csv` files.

The function `strsplit()` splits the elements of a character vector.

The package *zoo* contains functions `write.zoo()` and `read.zoo()` for writing and reading *zoo* time series from `.txt` and `.csv` files.

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list.

The function `assign()` assigns a value to an object in a specified *environment*, by referencing it using a character string (name).

The function `save()` writes objects to compressed binary `.RData` files.

```
> # Column names end with ".Close"
> colnames(pricev)
> strsplit(colnames(pricev), split="[.]")
> do.call(rbind, strsplit(colnames(pricev), split="[.]"))
> do.call(rbind, strsplit(colnames(pricev), split="[.]"))[, 1]
> # Drop ".Close" from colnames
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> tail(pricev, 3)
> # Which objects in global environment are class xts?
> unlist(eapply(globalenv(), is.xts))
> # Save xts to csv file
> write.zoo(pricev,
+   file="/Users/jerzy/Develop/lecture_slides/data/etf_series.csv",
> # Copy prices into etfenv
> etfenv$prices <- pricev
> # Or
> assign("pricev", pricev, envir=etfenv)
> # Save to .RData file
> save(etfenv, file="etf_data.RData")
```

Calculating Percentage Returns from Close Prices

The function quantmod::dailyReturn() calculates the percentage daily returns from the *Close* prices.

The lapply() and sapply() functionals perform a loop over the columns of *zoo* and *xts* series.

```
> # Extract VTI prices
> pricev <- etfenv$prices[ ,"VTI"]
> pricev <- na.omit(pricev)
> # Calculate percentage returns "by hand"
> pricel <- as.numeric(pricev)
> pricel <- c(pricel[1], pricel[-NROW(pricel)])
> pricel <- xts(pricel, zoo::index(pricev))
> retp <- (pricev-pricel)/pricel
> # Calculate percentage returns using dailyReturn()
> retd <- quantmod::dailyReturn(pricev)
> head(cbind(retd, retp))
> all.equal(retd, retp, check.attributes=FALSE)
> # Calculate returns for all prices in etfenv$prices
> retp <- lapply(etfenv$prices, function(xtsv) {
+   retd <- quantmod::dailyReturn(na.omit(xtsv))
+   colnames(retd) <- names(xtsv)
+   retd
+ })  # end lapply
> # "retp" is a list of xts
> class(retp)
> class(retp[[1]])
> # Flatten list of xts into a single xts
> retp <- do.call(cbind, retp)
> class(retp)
> dim(retp)
> # Copy retp into etfenv and save to .RData file
> # assign("retp", retp, envir=etfenv)
> etfenv$retp <- retp
> save(etfenv, file="/Users/jerzy/Develop/lecture_slides/data/etf_da
```

Managing Data Inside Environments

The function `as.environment()` coerces objects (listv) into an environment.

The function `eapply()` is similar to `lapply()`, and applies a function to objects in an *environment*, and returns a list.

The function `mget()` accepts a vector of strings and returns a list of the corresponding objects extracted from an *environment*.

```
> library(rutils)
> startd <- "2012-05-10"; endd <- "2013-11-20"
> # Select all objects in environment and return as environment
> newenv <- as.environment(eapply(etfenv, "[",
+             paste(startd, endd, sep="/")))
> # Select only symbolv in environment and return as environment
> newenv <- as.environment(
+   lapply(as.list(etfenv)[symbolv], "[",
+    paste(startd, endd, sep="/")))
> # Extract and cbind Close prices and return to environment
> assign("prices", rutils::do_call(cbind,
+   lapply(ls(etfenv), function(symboln) {
+     xtsv <- quantmod::Cl(get(symboln, etfenv))
+     colnames(xtsv) <- symboln
+     xtsv
+   })), envir=newenv)
> # Get sizes of OHLC xts series in etfenv
> sapply(mget(symbolv, envir=etfenv), object.size)
> # Extract and cbind adjusted prices and return to environment
> colname <- function(xtsv)
+   strsplit(colnames(xtsv), split="[.]")[[1]][1]
> assign("prices", rutils::do_call(cbind,
+          lapply(mget(etfenv$symbolv, envir=etfenv),
+                 function(xtsv) {
+                   xtsv <- Ad(xtsv)
+                   colnames(xtsv) <- colname(xtsv)
+                   xtsv
+          })), envir=newenv)
```

Stock Databases And Survivorship Bias

The file sp500_constituents.csv contains a *data frame* of over 700 present (and also some past) *S&P500* index constituents.

The file sp500_constituents.csv is updated with stocks recently added to the *S&P500* index by downloading the *SPY ETF Holdings*.

But the file sp500_constituents.csv doesn't include companies that have gone bankrupt. For example, it doesn't include Enron, which was in the *S&P500* index before it went bankrupt in 2001.

Most databases of stock prices don't include companies that have gone bankrupt or have been liquidated.

This introduces a *survivorship bias* to the data, which can skew portfolio simulations and strategy backtests.

Accurate strategy simulations require starting with a portfolio of companies at a "point in time" in the past, and tracking them over time.

Research databases like the *WRDS* database provide stock prices of companies that are no longer traded.

The stock tickers are stored in the column "Ticker" of the sp500 *data frame*.

Some tickers (like "BRK.B" and "BF.B") are not valid symbols in *Tiingo*, so they must be renamed.

```
> # Load data frame of S&P500 constituents from CSV file
> sp500 <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/s
> # Inspect data frame of S&P500 constituents
> dim(sp500)
> colnames(sp500)
> # Extract tickers from the column Ticker
> symbolv <- sp500$Ticker
> # Get duplicate tickers
> tablev <- table(symbolv)
> duplicatv <- tablev[tablev > 1]
> duplicatv <- names(duplicatv)
> # Get duplicate records (rows) of sp500
> sp500[symbolv %in% duplicatv, ]
> # Get unique tickers
> symbolv <- unique(symbolv)
> # Find index of ticker "BRK.B"
> which(symbolv=="BRK.B")
> # Rename "BRK.B" to "BRK-B" and "BF.B" to "BF-B"
> symbolv[which(symbolv=="BRK.B")] <- "BRK-B"
> symbolv[which(symbolv=="BF.B")] <- "BF-B"
```

Downloading Stock Time Series From *Tiingo*

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Tiingo*, *Alpha Vantage*, and Quandl as the only major providers of free daily *OHLC* stock prices.

But Quandl doesn't provide free *ETF* prices, while *Tiingo* does.

The function `getSymbols()` has a *method* for downloading time series data from *Tiingo*, called `getSymbols.tiingo()`.

Users must first obtain a *Tiingo API key*, and then pass it in `getSymbols.tiingo()` calls:
https://www.tiingo.com/

Note that the data are downloaded as `xts` time series, with a date-time index of class `POSIXct` (not `Date`).

```
> # Load package rutils
> library(rutils)
> # Create new environment for data
> sp500env <- new.env()
> # Boolean vector of symbols already downloaded
> isdownloaded <- symbolv %in% ls(sp500env)
> # Download in while loop from Tiingo and copy into environment
> nattempts <- 0  # Number of download attempts
> while ((sum(!isdownloaded) > 0) & (nattempts<3)) {
+   # Download data and copy it into environment
+   nattempts <- nattempts + 1
+   cat("Download attempt = ", nattempts, "\n")
+   for (symboln in symbolv[!isdownloaded]) {
+     cat("processing: ", symboln, "\n")
+     tryCatch(  # With error handler
+ quantmod::getSymbols(symboln, src="tiingo", adjust=TRUE, auto.assi
+            from="1990-01-01", env=sp500env, api.key="j84ac2b9c5bde
+ # Error handler captures error condition
+ error=function(msg) {
+   print(paste0("Error handler: ", msg))
+ },  # end error handler
+ finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdownloaded <- symbolv %in% ls(sp500env)
+   Sys.sleep(2)  # Wait 2 seconds until next attempt
+ }  # end while
> class(sp500env$AAPL)
> class(zoo::index(sp500env$AAPL))
> tail(sp500env$AAPL)
> symbolv[!isdownloaded]
```

Coercing Date-time Indices

The date-time indices of the *OHLC* stock prices are in the POSIXct format suitable for intraday prices, not daily prices.

The function as.Date() coerces POSIXct objects into Date objects.

The function get() retrieves objects that are referenced using character strings, instead of their names.

The function assign() assigns a value to an object in a specified *environment*, by referencing it using a character string (name).

The functions get() and assign() allow retrieving and assigning values to objects that are referenced using character strings.

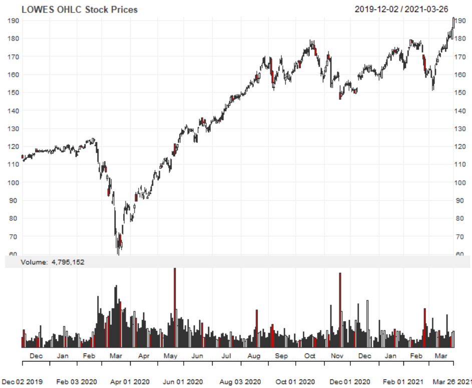
```
> # The date-time index of AAPL is POSIXct
> class(zoo::index(sp500env$AAPL))
> # Coerce the date-time index of AAPL to Date
> zoo::index(sp500env$AAPL) <- as.Date(zoo::index(sp500env$AAPL))
> # Coerce all the date-time indices to Date
> for (symboln in ls(sp500env)) {
+   ohlc <- get(symboln, envir=sp500env)
+   zoo::index(ohlc) <- as.Date(zoo::index(ohlc))
+   assign(symboln, ohlc, envir=sp500env)
+ }  # end for
```

Managing Exceptions in Stock Symbols

The column names for symbol "LOW" (Lowe's company) must be renamed for the extractor function quantmod::Lo() to work properly.

Tickers which contain a dot in their name (like "BRK.B") are not valid symbols in R, so they must be downloaded separately and renamed.

```
> # "LOW.Low" is a bad column name
> colnames(sp500env$LOW)
> strsplit(colnames(sp500env$LOW), split="[.]")
> do.call(cbind, strsplit(colnames(sp500env$LOW), split="[.]"))
> do.call(cbind, strsplit(colnames(sp500env$LOW), split="[.]"))[2, ]
> # Extract proper names from column names
> namev <- rutils::get_name(colnames(sp500env$LOW), field=2)
> # Or
> # namev <- do.call(rbind, strsplit(colnames(sp500env$LOW),
> #                               split="[.]"))[, 2]
> # Rename "LOW" colnames to "LOWES"
> colnames(sp500env$LOW) <- paste("LOVES", namev, sep=".")
> sp500env$LOWES <- sp500env$LOW
> rm(LOW, envir=sp500env)
> # Rename BF-B colnames to "BFB"
> colnames(sp500env$"BF-B") <- paste("BFB", namev, sep=".")
> sp500env$BFB <- sp500env$"BF-B"
> rm("BF-B", envir=sp500env)
> # Rename BRK-B colnames
> sp500env$BRKB <- sp500env$`BRK-B`
> rm(`BRK-B`, envir=sp500env)
> colnames(sp500env$BRKB) <- gsub("BRK-B", "BRKB", colnames(sp500en
> # Save OHLC prices to .RData file
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> # Download "BRK.B" separately with auto.assign=FALSE
> # BRKB <- quantmod::getSymbols("BRK-B", auto.assign=FALSE, src="tiingo", adjust=TRUE, from="1990-01-01", api.key="j84ac2b9c5bde2d68e3
> # colnames(BRKB) <- paste("BRKB", namev, sep=".")
> # sp500env$BRKB <- BRKB
```



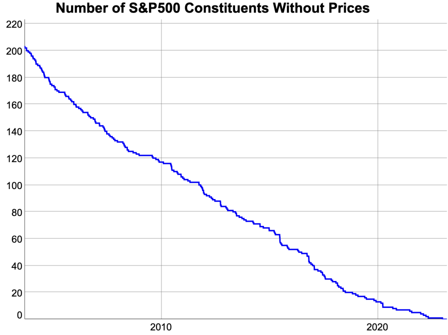

```
> # Plot OHLC candlestick chart for LOWES
> chart_Series(x=sp500env$LOWES["2019-12/"],
+   TA="add_Vo()", name="LOWES OHLC Stock Prices")
> # Plot dygraph
> dygraphs::dygraph(sp500env$LOWES["2019-12/", -5], main="LOWES OHL
+   dyCandlestick()
```

S&P500 Stock Index Constituent Prices

The file `sp500.RData` contains the *environment* `sp500_env` with *OHLC* prices and trading volumes of *S&P500* stock index constituents.

The *S&P500* stock index constituent data is of poor quality before 2000, so we'll mostly use the data after the year 2000.

```
> # Load S&P500 constituent stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> pricev <- eapply(sp500env, quantmod::Cl)
> pricev <- rutils::do_call(cbind, pricev)
> # Carry forward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> # Drop ".Close" from column names
> colnames(pricev)
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> # Calculate percentage returns of the S&P500 constituent stocks
> # retp <- xts::diff.xts(log(pricev))
> retp <- xts::diff.xts(pricev)/
+   rutils::lagit(pricev, pad_zeros=FALSE)
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> samplev <- sample(NCOL(retp), s=100, replace=FALSE)
> prices100 <- pricev[, samplev]
> returns100 <- retp[, samplev]
> save(pricev, prices100,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_prices.RData")
> save(retp, returns100,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.RData")
```

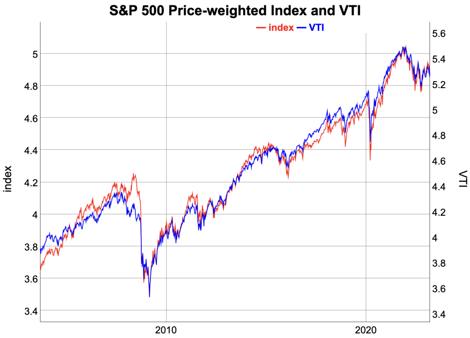



```
> # Calculate number of constituents without prices
> datav <- rowSums(is.na(pricev))
> datav <- xts::xts(datav, order.by=zoo::index(pricev))
> dygraphs::dygraph(datav, main="Number of S&P500 Constituents With
+   dyOptions(colors="blue", strokeWidth=2)
```

S&P500 Stock Portfolio Index

The price-weighted index of *S&P500* constituents closely follows the VTI *ETF*.

```
> # Calculate price weighted index of constituent
> ncols <- NCOL(pricev)
> pricev <- zoo::na.locf(pricev, fromLast=TRUE)
> indeks <- xts(rowSums(pricev)/ncols, zoo::index(pricev))
> colnames(indeks) <- "index"
> # Combine index with VTI
> datav <- cbind(indeks[zoo::index(etfenv$VTI)], etfenv$VTI[, 4])
> colnamev <- c("index", "VTI")
> colnames(datav) <- colnamev
> # Plot index with VTI
> endd <- rutils::calc_endpoints(datav, interval="weeks")
> dygraphs::dygraph(log(datav)[endd],
+   main="S&P 500 Price-weighted Index and VTI") %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", col="red") %>%
+   dySeries(name=colnamev[2], axis="y2", col="blue")
```



Writing Time Series To Files

The data from *Tiingo* is downloaded as xts time series, with a date-time index of class POSIXct (not Date).

The function save() writes objects to compressed binary .RData files.

The easiest way to share data between R and Excel is through .csv files.

The package *zoo* contains functions write.zoo() and read.zoo() for writing and reading *zoo* time series from .txt and .csv files.

The function data.table::fread() reads from .csv files over 6 times faster than the function read.csv()!

The function data.table::fwrite() writes to .csv files over 12 times faster than the function write.csv(), and 278 times faster than function cat()!

```
> # Save the environment to compressed .RData file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> save(sp500env, file=paste0(dirn, "sp500.RData"))
> # Save the ETF prices into CSV files
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/SP500/"
> for (symboln in ls(sp500env)) {
+   zoo::write.zoo(sp500env$symbol, file=paste0(dirn, symboln, ".csv
+ }  # end for
> # Or using lapply()
> filens <- lapply(ls(sp500env), function(symboln) {
+   xtsv <- get(symboln, envir=sp500env)
+   zoo::write.zoo(xtsv, file=paste0(dirn, symboln, ".csv"))
+   symboln
+ })  # end lapply
> unlist(filens)
> # Or using eapply() and data.table::fwrite()
> filens <- eapply(sp500env , function(xtsv) {
+   filen <- rutils::get_name(colnames(xtsv)[1])
+   data.table::fwrite(data.table::as.data.table(xtsv), file=paste0(
+   filen
+ })  # end eapply
> unlist(filens)
```

Reading Time Series from Files

The function load() reads data from .RData files, and *invisibly* returns a vector of names of objects created in the workspace.

The function Sys.glob() listv files matching names obtained from wildcard expansion.

The easiest way to share data between R and Excel is through .csv files.

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

The function data.table::setDF() coerces a *data table* object into a *data frame* using a *side effect*, without making copies of data.

The function data.table::fread() reads from .csv files over 6 times faster than the function read.csv()!

```
> # Load the environment from compressed .RData file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> load(file=paste0(dirn, "sp500.RData"))
> # Get all the .csv file names in the directory
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/SP500/"
> filens <- Sys.glob(paste0(dirn, "*.csv"))
> # Create new environment for data
> sp500env <- new.env()
> for (filen in filens) {
+   xtsv <- xts::as.xts(zoo::read.csv.zoo(filen))
+   symboln <- rutils::get_name(colnames(xtsv)[1])
+   # symboln <- strsplit(colnames(xtsv), split="[.]")[[1]][1]
+   assign(symboln, xtsv, envir=sp500env)
+ }  # end for
> # Or using fread()
> for (filen in filens) {
+   xtsv <- data.table::fread(filen)
+   data.table::setDF(xtsv)
+   xtsv <- xts::xts(xtsv[, -1], as.Date(xtsv[, 1]))
+   symboln <- rutils::get_name(colnames(xtsv)[1])
+   assign(symboln, xtsv, envir=sp500env)
+ }  # end for
```

Downloading Stock Time Series From *Alpha Vantage*

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Tiingo*, *Alpha Vantage*, and Quandl as the only major providers of free daily *OHLC* stock prices.

But Quandl doesn't provide free *ETF* prices, while *Alpha Vantage* does.

The function `getSymbols()` has a *method* for downloading time series data from *Alpha Vantage*, called `getSymbols.av()`.

Users must first obtain an *Alpha Vantage API key*, and then pass it in `getSymbols.av()` calls:
https://www.alphavantage.co/

The function `adjustOHLC()` with argument `use.Adjusted=TRUE`, adjusts all the *OHLC* price columns, using the *Adjusted* price column.

```
> # Remove all files from environment(if necessary)
> rm(list=ls(sp500env), envir=sp500env)
> # Download in while loop from Alpha Vantage and copy into environm
> isdownloaded <- symbolv %in% ls(sp500env)
> nattempts <- 0
> while ((sum(!isdownloaded) > 0) & (nattempts<10)) {
+   # Download data and copy it into environment
+   nattempts <- nattempts + 1
+   for (symboln in symbolv[!isdownloaded]) {
+     cat("processing: ", symboln, "\n")
+     tryCatch(  # With error handler
+ quantmod::getSymbols(symboln, src="av", adjust=TRUE, auto.assign=T
+            output.size="full", api.key="T7JPW54ES8G75310"),
+ # error handler captures error condition
+ error=function(msg) {
+   print(paste0("Error handler: ", msg))
+ },  # end error handler
+ finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdownloaded <- symbolv %in% ls(sp500env)
+   Sys.sleep(2)  # Wait 2 seconds until next attempt
+ }  # end while
> # Adjust all OHLC prices in environment
> for (symboln in ls(sp500env)) {
+   assign(symboln,
+     adjustOHLC(get(x=symboln, envir=sp500env), use.Adjusted=TRUE),
+     envir=sp500env)
+ }  # end for
```

Downloading The *S&P500* Index Time Series From *Yahoo*

The *S&P500* stock market index is a capitalization-weighted average of the 500 largest U.S. companies, and covers about 80% of the U.S. stock market capitalization.

Notice: *Yahoo* no longer provides a public API for data.

There are workarounds but they're tedious.

Yahoo provides daily *OHLC* prices for the *S&P500* index (symbol *^GSPC*), and for the *S&P500* total return index (symbol *^SP500TR*).

But special characters in some stock symbols, like `"-"` or `"^"` are not allowed in `R` names.

For example, the symbol *^GSPC* for the *S&P500* stock market index isn't a valid name in `R`.

The function `setSymbolLookup()` creates valid names corresponding to stock symbols, which are then used by the function `getSymbols()` to create objects with the valid names.

Yahoo data quality deteriorated significantly in 2017, and *Google* data quality is also poor, leaving *Alpha Vantage* and Quandl as the only major providers of free daily *OHLC* stock prices.

```
> # Assign name SP500 to ^GSPC symbol
> quantmod::setSymbolLookup(SP500=list(name="^GSPC", src="yahoo"))
> quantmod::getSymbolLookup()
> # View and clear options
> options("getSymbols.sources")
> options(getSymbols.sources=NULL)
> # Download S&P500 prices into etfenv
> quantmod::getSymbols("SP500", env=etfenv,
+     adjust=TRUE, auto.assign=TRUE, from="1990-01-01")
>
> chart_Series(x=etfenv$SP500["2016/"],
+        TA="add_Vo()", name="S&P500 index")
```

Downloading The *DJIA* Index Time Series From *Yahoo*

The Dow Jones Industrial Average (*DJIA*) stock market index is a price-weighted average of the 30 largest U.S. companies (same number of shares per company).

Yahoo provides daily *OHLC* prices for the *DJIA* index (symbol *^DJI*), and for the *DJITR* total return index (symbol *DJITR*).

But special characters in some stock symbols, like "-" or "^" are not allowed in R names.

For example, the symbol *^DJI* for the *DJIA* stock market index isn't a valid name in R.

The function `setSymbolLookup()` creates valid names corresponding to stock symbols, which are then used by the function `getSymbols()` to create objects with the valid names.

```
> # Assign name DJIA to ^DJI symbol
> setSymbolLookup(DJIA=list(name="^DJI", src="yahoo"))
> getSymbolLookup()
> # view and clear options
> options("getSymbols.sources")
> options(getSymbols.sources=NULL)
> # Download DJIA prices into etfenv
> quantmod::getSymbols("DJIA", env=etfenv,
+     adjust=TRUE, auto.assign=TRUE, from="1990-01-01")
> chart_Series(x=etfenv$DJIA["2016/"],
+        TA="add_Vo()", name="DJIA index")
```

Calculating Prices and Returns From *OHLC* Data

The function `na.locf()` from package *zoo* replaces `NA` values with the most recent non-`NA` values prior to it.

The function `na.locf()` with argument `fromLast=TRUE` replaces `NA` values with non-`NA` values in reverse order, starting from the end.

The function `rutils::get_name()` extracts symbol names (tickers) from a vector of character strings.

```
> pricev <- eapply(sp500env, quantmod::Cl)
> pricev <- rutils::do_call(cbind, pricev)
> # Carry forward non-NA prices
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> # Get first column name
> colnames(pricev[, 1])
> rutils::get_name(colnames(pricev[, 1]))
> # Modify column names
> colnames(pricev) <- rutils::get_name(colnames(pricev))
> # Or
> # colnames(pricev) <- do.call(rbind,
> #   strsplit(colnames(pricev), split="[.]"))[, 1]
> # Calculate percentage returns
> retp <- xts::diff.xts(pricev)/
+   rutils::lagit(pricev, pad_zeros=FALSE)
> # Select a random sample of 100 prices and returns
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> samplev <- sample(NCOL(retp), s=100, replace=FALSE)
> prices100 <- pricev[, samplev]
> returns100 <- retp[, samplev]
> # Save the data into binary files
> save(pricev, prices100,
+      file="/Users/jerzy/Develop/lecture_slides/data/sp500_prices.R
> save(retp, returns100,
+      file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns.
```

Downloading Stock Prices From Polygon

Polygon is a premium provider of live and historical stock price data, both daily and intraday (minutes).

Polygon provides 2 years of daily historical stock prices for free. But users must first obtain a Polygon *API key*.

Polygon provides the historical *OHLC* stock prices in *JSON* format.

JSON (JavaScript Object Notation) is a data format consisting of symbol-value pairs.

The package *jsonlite* contains functions for managing data in *JSON* format.

The functions `fromJSON()` and `toJSON()` convert data from *JSON* format to `R` objects, and vice versa.

The functions `read_json()` and `write_json()` read and write *JSON* format data in files.

The function `download.file()` downloads data from an internet website `URL` and writes it to a file.

```
> # Setup code
> symboln <- "SPY"
> startd <- as.Date("1990-01-01")
> todayd <- Sys.Date()
> tspan <- "day"
> # Replace below your own Polygon API key
> apikey <- "SEpnsBpiRyONMJdl48r6dOo0_pjmCu5r"
> # Create url for download
> urll <- paste0("https://api.polygon.io/v2/aggs/ticker/", symboln,
> # Download SPY OHLC prices in JSON format from Polygon
> ohlc <- jsonlite::read_json(urll)
> class(ohlc)
> NROW(ohlc)
> names(ohlc)
> # Extract list of prices from json object
> ohlc <- ohlc$results
> # Coerce from list to matrix
> ohlc <- lapply(ohlc, unlist)
> ohlc <- do.call(rbind, ohlc)
> # Coerce time from milliseconds to dates
> datev <- ohlc[, "t"]/1e3
> datev <- as.POSIXct(datev, origin="1970-01-01")
> datev <- as.Date(datev)
> tail(datev)
> # Coerce from matrix to xts
> ohlc <- ohlc[, c("o","h","l","c","v","vw")]
> colnames(ohlc) <- c("Open", "High", "Low", "Close", "Volume", "VWA
> ohlc <- xts::xts(ohlc, order.by=datev)
> tail(ohlc)
> # Save the xts time series to compressed RData file
> save(ohlc, file="/Users/jerzy/Data/spy_daily.RData")
> # Candlestick plot of SPY OHLC prices
> dygraphs::dygraph(ohlc[, 1:4], main=paste("Candlestick Plot of", s
+   dygraphs::dyCandlestick()
```

Downloading Multiple Stock Prices From Polygon

The stock prices for multiple stocks can be downloaded in a `while()` loop.

```
> # Select ETF symbols for asset allocation
> symbolv <- c("VTI", "VEU", "EEM", "XLY", "XLP", "XLI
+  "XLV", "XLI", "XLB", "XLK", "XLU", "VYM", "IVW", "I
+  "IWF", "IEF", "TLT", "VNQ", "DBC", "GLD", "USO", "
+  "MTUM", "IVE", "VLUE", "QUAL", "VTV", "USMV", "AIE
> # Setup code
> etfenv <- new.env()  # New environment for data
> # Boolean vector of symbols already downloaded
> isdownloaded <- symbolv %in% ls(etfenv)
```

```
> # Download data from Polygon using while loop
> while (sum(!isdownloaded) > 0) {
+   for (symboln in symbolv[!isdownloaded]) {
+     cat("Processing:", symboln, "\n")
+     tryCatch({  # With error handler
+ # Download OHLC bars from Polygon into JSON format file
+ urll <- paste0("https://api.polygon.io/v2/aggs/ticker/", symboln, "/range/1/",
+ ohlc <- jsonlite::read_json(urll)
+ # Extract list of prices from json object
+ ohlc <- ohlc$results
+ # Coerce from list to matrix
+ ohlc <- lapply(ohlc, unlist)
+ ohlc <- do.call(rbind, ohlc)
+ # Coerce time from milliseconds to dates
+ datev <- ohlc[, "t"]/1e3
+ datev <- as.POSIXct(datev, origin="1970-01-01")
+ datev <- as.Date(datev)
+ # Coerce from matrix to xts
+ ohlc <- ohlc[, c("o","h","l","c","v","vw")]
+ colnames(ohlc) <- paste0(symboln, ".", c("Open", "High", "Low", "Close", "Volu
+ ohlc <- xts::xts(ohlc, order.by=datev)
+ # Save to environment
+ assign(symboln, ohlc, envir=etfenv)
+ Sys.sleep(1)
+ },
+     error={function(msg) print(paste0("Error handler: ", msg))},
+     finally=print(paste0("Symbol = ", symboln))
+     )  # end tryCatch
+   }  # end for
+   # Update vector of symbols already downloaded
+   isdownloaded <- symbolv %in% ls(etfenv)
+ }  # end while
> save(etfenv, file="/Users/jerzy/Develop/lecture_slides/data/etf_data.RData")
```

Calculating the Stock Alphas, Betas, and Other Performance Statistics

The package *PerformanceAnalytics* contains functions for calculating risk and performance statistics, such as the *variance*, *skewness*, *kurtosis*, *beta*, *alpha*, etc.

The function `PerformanceAnalytics::table.CAPM()` calculates the *beta* β and *alpha* α values, the *Treynor* ratio, and other performance statistics.

The function `PerformanceAnalytics::table.Stats()` calculates a data frame of risk and return statistics of the return distributions.

```
> prices <- eapply(etfenv, quantmod::Cl)
> prices <- do.call(cbind, prices)
> # Drop ".Close" from colnames
> colnames(prices) <- do.call(rbind, strsplit(colnames(prices), spli
> # Calculate the log returns
> retp <- xts::diff.xts(log(prices))
> # Copy prices and returns into etfenv
> etfenv$prices <- prices
> etfenv$retp <- retp
> # Copy symbolv into etfenv
> etfenv$symbolv <- symbolv
> # Calculate the risk-return statistics
> riskstats <- PerformanceAnalytics::table.Stats(retp)
> # Transpose the data frame
> riskstats <- as.data.frame(t(riskstats))
> # Add Name column
> riskstats$Name <- rownames(riskstats)
> # Copy riskstats into etfenv
> etfenv$riskstats <- riskstats
> # Calculate the beta, alpha, Treynor ratio, and other performance
> capmstats <- PerformanceAnalytics::table.CAPM(Ra=retp[, symbolv],
+                                               Rb=retp[, "VTI"], scale=2
> colnamev <- strsplit(colnames(capmstats), split=" ")
> colnamev <- do.call(cbind, colnamev)[1, ]
> colnames(capmstats) <- colnamev
> capmstats <- t(capmstats)
> capmstats <- capmstats[, -1]
> colnamev <- colnames(capmstats)
> whichv <- match(c("Annualized Alpha", "Information Ratio", "Treyno
> colnamev[whichv] <- c("Alpha", "Information", "Treynor")
> colnames(capmstats) <- colnamev
> capmstats <- capmstats[order(capmstats[, "Alpha"], decreasing=TRUE
> # Copy capmstats into etfenv
> etfenv$capmstats <- capmstats
> save(etfenv, file="/Users/jerzy/Develop/lecture_slides/data/etf_da
```

Scraping *S&P500* Stock Index Constituents From Websites

The *S&P500* index constituents change over time, and *Standard & Poor's* replaces companies that have decreased in capitalization with ones that have increased.

The *S&P500* index may contain more than 500 stocks because some companies have several share classes of stock.

The *S&P500* index constituents may be scraped from websites like Wikipedia, using dedicated packages.

The function `getURL()` from package *RCurl* downloads the *html* text data from an internet website URL.

The function `readHTMLTable()` from package *XML* extracts tables from *html* text data or from a remote URL, and returns them as a list of *data frames* or matrices.

`readHTMLTable()` can't parse secure URLs, so they must first be downloaded using function `getURL()`, and then parsed using `readHTMLTable()`.

```
> library(RCurl)  # Load package RCurl
> library(XML)  # Load package XML
> # Download text data from URL
> sp500 <- getURL(
+   "https://en.wikipedia.org/wiki/List_of_S%26P500_companies")
> # Extract tables from the text data
> sp500 <- readHTMLTable(sp500)
> str(sp500)
> # Extract colnames of data frames
> lapply(sp500, colnames)
> # Extract S&P500 constituents
> sp500 <- sp500[[1]]
> head(sp500)
> # Create valid R names from symbols containing "-" or "."characte
> sp500$namev <- gsub("-", "_", sp500$Ticker)
> sp500$namev <- gsub("[.]", "_", sp500$names)
> # Write data frame of S&P500 constituents to CSV file
> write.csv(sp500,
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_Yahoo.csv",
+   row.names=FALSE)
```

Downloading *S&P500* Time Series Data From *Yahoo*

Before time series data for the *S&P500* index constituents can be downloaded from *Yahoo*, it's necessary to create valid names corresponding to symbols containing special characters like `"-"`.

The function `setSymbolLookup()` creates a lookup table for *Yahoo* symbols, using valid names in `R`.

For example *Yahoo* uses the symbol `"BRK-B"`, which isn't a valid name in `R`, but can be mapped to `"BRK_B"`, using the function `setSymbolLookup()`.

```
> library(rutils)  # Load package rutils
> # Load data frame of S&P500 constituents from CSV file
> sp500 <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/s
> # Register symbols corresponding to R names
> for (indeks in 1:NROW(sp500)) {
+   cat("processing: ", sp500$Ticker[indeks], "\n")
+   setSymbolLookup(structure(
+     list(list(name=sp500$Ticker[indeks])),
+     names=sp500$names[indeks]))
+ }  # end for
> sp500env <- new.env()  # new environment for data
> # Remove all files (if necessary)
> rm(list=ls(sp500env), envir=sp500env)
> # Download data and copy it into environment
> rutils::get_data(sp500$names,
+    env_out=sp500env, startd="1990-01-01")
> # Or download in loop
> for (symboln in sp500$names) {
+   cat("processing: ", symboln, "\n")
+   rutils::get_data(symboln,
+    env_out=sp500env, startd="1990-01-01")
+ }  # end for
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp50
> chart_Series(x=sp500env$BRKB["2016/"],
+        TA="add_Vo()", name="BRK-B stock")
```

Downloading *FRED* Time Series Data

FRED is a database of economic time series maintained by the Federal Reserve Bank of St. Louis:
http://research.stlouisfed.org/fred2/

The function `getSymbols()` downloads time series data into the specified *environment*.

`getSymbols()` can download *FRED* data with the argument `"src"` set to `FRED`.

If the argument `"auto.assign"` is set to `FALSE`, then `getSymbols()` returns the data, instead of assigning it silently.


```
> # Download U.S. unemployment rate data
> unrate <- quantmod::getSymbols("UNRATE",
+    auto.assign=FALSE, src="FRED")
> # Plot U.S. unemployment rate data
> dygraphs::dygraph(unrate["1990/"], main="U.S. Unemployment Rate")
+   dyOptions(colors="blue", strokeWidth=2)
> # Or
> quantmod::chart_Series(unrate["1990/"], name="U.S. Unemployment Ra
```

The *Quandl* Database

Quandl is a distributor of third party data, and offers several million financial, economic, and social datasets.

Much of the Quandl data is free, while premium data can be obtained under a temporary license.

Quandl provides online help and a guide to its datasets:
https://www.quandl.com/help/r
https://www.quandl.com/browse
https://www.quandl.com/blog/getting-started-with-the-quandl-api
https://www.quandl.com/blog/stock-market-data-guide

Quandl provides stock prices, stock fundamentals, financial ratios, zoo::indexes, options and volatility, earnings estimates, analyst ratings, etc.:
https://www.quandl.com/blog/api-for-stock-data

```
> install.packages("devtools")
> library(devtools)
> # Install package Quandl from github
> install_github("quandl/R-package")
> library(Quandl)  # Load package Quandl
> # Register Quandl API key
> Quandl.api_key("pVJi9Nv3V8CD3Js5s7Qx")
> # Get short description
> packageDescription("Quandl")
> # Load help page
> help(package="Quandl")
> # Remove Quandl from search path
> detach("package:Quandl")
```

Quandl has developed an R package called *Quandl* that allows downloading data from Quandl directly into R.

To make more than 50 downloads a day, you need to register your *Quandl API key* using the function `Quandl.api_key()`,

Downloading Time Series Data from *Quandl*

Quandl data can be downloaded directly into R using the function Quandl().

The dots "..." argument of the Quandl() function accepts additional parameters to the *Quandl API*,

Quandl datasets have a unique *Quandl code* in the format "database/ticker", which can be found on the Quandl website for that dataset:
https://www.quandl.com/data/WIKI?keyword=aapl

WIKI is a user maintained free database of daily prices for 3,000 U.S. stocks,
https://www.quandl.com/data/WIKI

SEC is a free database of stock fundamentals extracted from *SEC 10Q* and *10K* filings (but not harmonized),
https://www.quandl.com/data/SEC

RAYMOND is a free database of harmonized stock fundamentals, based on the *SEC* database,
https://www.quandl.com/data/RAYMOND-Raymond https://www.quandl.com/data/RAYMOND-Raymond?keyword=aapl

```
> library(rutils)  # Load package rutils
> # Download EOD AAPL prices from WIKI free database
> pricev <- Quandl(code="WIKI/AAPL",
+   type="xts", startd="1990-01-01")
> x11(width=14, height=7)
> chart_Series(pricev["2016", 1:4], name="AAPL OHLC prices")
> # Add trade volume in extra panel
> add_TA(pricev["2016", 5])
> # Download euro currency rates
> pricev <- Quandl(code="BNP/USDEUR",
+     startd="2013-01-01",
+     endd="2013-12-01", type="xts")
> # Download multiple time series
> pricev <- Quandl(code=c("NSE/OIL", "WIKI/AAPL"),
+     startd="2013-01-01", type="xts")
> # Download AAPL gross profits
> prof_it <- Quandl("RAYMOND/AAPL_GROSS_PROFIT_Q", type="xts")
> chart_Series(prof_it, name="AAPL gross profits")
> # Download Hurst time series
> pricev <- Quandl(code="PE/AAPL_HURST",
+     startd="2013-01-01", type="xts")
> chart_Series(pricev["2016/", 1], name="AAPL Hurst")
```

Stock Index and Instrument Metadata on *Quandl*

Instrument metadata specifies properties of instruments, like its currency, contract size, tick value, delievery months, start date, etc.

Quandl provides instrument metadata for stock indices, futures, and currencies:

https://www.quandl.com/blog/useful-listv

Quandl also provides constituents for stock indices, for example the *S&P500*, *Dow Jones Industrial Average*, *NASDAQ Composite*, *FTSE 100*, etc.

```
> # Load S&P500 stock Quandl codes
> sp500 <- read.csv(
+   file="/Users/jerzy/Develop/lecture_slides/data/sp500_quandl.csv'
> # Replace "-" with "_" in symbols
> sp500$free_code <- gsub("-", "_", sp500$free_code)
> head(sp500)
> # vector of symbols in sp500 frame
> tickers <- gsub("-", "_", sp500$ticker)
> # Or
> tickers <- matrix(unlist(
+   strsplit(sp500$free_code, split="/"),
+   use.names=FALSE), ncol=2, byrow=TRUE)[, 2]
> # Or
> tickers <- do_call_rbind(
+   strsplit(sp500$free_code, split="/"))[, 2]
```

Downloading Multiple Time Series from *Quandl*

Time series data for a portfolio of stocks can be downloaded by performing a loop over the function `Quandl()` from package Quandl.

The `assign()` function assigns a value to an object in a specified *environment*, by referencing it using a character string (name).

```
> sp500env <- new.env()  # new environment for data
> # Remove all files (if necessary)
> rm(list=ls(sp500env), envir=sp500env)
> # Boolean vector of symbols already downloaded
> isdownloaded <- tickers %in% ls(sp500env)
> # Download data and copy it into environment
> for (ticker in tickers[!isdownloaded]) {
+   cat("processing: ", ticker, "\n")
+   datav <- Quandl(code=paste0("WIKI/", ticker),
+             startd="1990-01-01", type="xts")[, -(1:7)]
+   colnames(datav) <- paste(ticker,
+     c("Open", "High", "Low", "Close", "Volume"), sep=".")
+   assign(ticker, datav, envir=sp500env)
+ }  # end for
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp5
> chart_Series(x=sp500env$XOM["2016/"], TA="add_Vo()", name="XOM sto
```

Downloading Futures Time Series from *Quandl*

Quandl provides the Wiki CHRIS Database of time series of prices for 600 different futures contracts.

The Wiki CHRIS Database contains daily *OHLC* prices for continuous futures contracts.

A continuous futures contract is a time series of prices obtained by chaining together prices from consecutive futures contracts.

The data is curated by the Quandl community from data provided by the *CME*, *ICE*, *LIFFE*, and other exchanges.

The *Quandl codes* are specified as `CHRIS/{EXCHANGE}_{CODE}{DEPTH}`, where `{DEPTH}` is the depth of the chained contract.

The chained front month contracts have depth 1, the back month contracts have depth 2, etc.

The continuous front and back month contracts allow building continuous futures curves.

Quandl data can be downloaded directly into `R` using the function `Quandl()`.

```
> library(Quandl)
> # Register Quandl API key
> Quandl.api_key("pVJi9Nv3V8CD3Js5s7Qx")
> # Download E-mini S&P500 futures prices
> pricev <- Quandl(code="CHRIS/CME_ES1",
+   type="xts", startd="1990-01-01")
> pricev <- pricev[, c("Open", "High", "Low", "Last", "Volume")]
> colnames(pricev)[4] <- "Close"
> # Plot the prices
> x11(width=5, height=4)  # Open x11 for plotting
> chart_Series(x=pricev["2008-06/2009-06"],
+        TA="add_Vo()", name="S&P500 Futures")
> # Plot dygraph
> dygraphs::dygraph(pricev["2008-06/2009-06", -5],
+   main="S&P500 Futures") %>%
+   dyCandlestick()
```

For example, the *Quandl code* for the continuous *E-mini S&P500* front month futures is `CHRIS/CME_ES1`, while for the back month it's `CHRIS/CME_ES2`, for the second back month it's `CHRIS/CME_ES3`, etc.

The *Quandl code* for the *E-mini Oil* futures is `CHRIS/CME_QM1`, for the *E-mini euro FX* futures is `CHRIS/CME_E71`, etc.

Downloading *VIX* Futures Files from CBOE

The CFE (CBOE Futures Exchange) provides daily CBOE Historical Data for Volatility Futures, including the *VIX* futures.

The CBOE data incudes *OHLC* prices and also the *settlement* price (in column `"Settle"`).

The *settlement* price is usually defined as the weighted average price (*WAP*) or the midpoint price, and is different from the *Close* price.

The *settlement* price is used for calculating the daily *mark to market* (value) of the futures contract.

Futures exchanges require that counterparties exchange (settle) the *mark to market* value of the futures contract daily, to reduce counterparty default risk.

The function `download.file()` downloads files from the internet.

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

```
> # Read CBOE futures expiration dates
> datev <- read.csv(file="/Users/jerzy/Develop/lecture_slides/data/f
+   row.names=1)
> dirn <- "/Users/jerzy/Develop/data/vix_data"
> dir.create(dirn)
> symbolv <- rownames(datev)
> filens <- file.path(dirn, paste0(symbolv, ".csv"))
> log_file <- file.path(dirn, "log_file.txt")
> cboe_url <- "https://markets.cboe.com/us/futures/market_statistics
> urls <- paste0(cboe_url, datev[, 1])
> # Download files in loop
> for (it in seq_along(urls)) {
+     tryCatch(  # Warning and error handler
+   download.file(urls[it],
+           destfile=filens[it], quiet=TRUE),
+ # Warning handler captures warning condition
+ warning=function(msg) {
+   cat(paste0("Warning handler: ", msg, "\n"), file=log_file, apper
+ },  # end warning handler
+ # Error handler captures error condition
+ error=function(msg) {
+   cat(paste0("Error handler: ", msg, "\n"), append=TRUE)
+ },  # end error handler
+ finally=cat(paste0("Processing file name = ", filens[it], "\n"), a
+     )  # end tryCatch
+ }  # end for
```

Downloading *VIX* Futures Data Into an Environment

The function `quantmod::getSymbols()` with the parameter `src="cfe"` downloads CFE data into the specified *environment*. (But this requires first loading the package *qmao*.)

Currently `quantmod::getSymbols()` doesn't download the most recent data.

```
> # Create new environment for data
> vix_env <- new.env()
> # Download VIX data for the months 6, 7, and 8 in 2018
> library(qmao)
> quantmod::getSymbols("VX", Months=1:12,
+   Years=2018, src="cfe", auto.assign=TRUE, env=vix_env)
> # Or
> qmao::getSymbols.cfe(Symbols="VX",
+   Months=6:8, Years=2018, env=vix_env,
+   verbose=FALSE, auto.assign=TRUE)
> # Calculate the classes of all the objects
> # In the environment vix_env
> unlist(eapply(vix_env, function(x) {class(x)[1]}))
> class(vix_env$VX_M18)
> colnames(vix_env$VX_M18)
> # Save the data to a binary file called "vix_cboe.RData".
> save(vix_env,
+   file="/Users/jerzy/Develop/data/vix_data/vix_cboe.RData")
```

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