R Packages

FRE6871 & FRE7241, Spring 2025

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

May 12, 2025

R Packages

Types of R Packages

R can run libraries of functions called packages,

R packages can can also contain data,

Most packages need to be *loaded* into R before they can be used,

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

There's also a special package called the base package, which is responsible for all the basic R functionality,

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

The *base* Packages

R includes a number of packages that are pre-installed (often called *base* packages),
Some *base* packages:

- *base* - basic R functionality,
- *stats* - statistical functions and random number generation,
- *graphics* - basic graphics,
- *utils* - utility functions,
- *datasets* - popular datasets,
- *parallel* - support for parallel computation,

Very popular packages:

- *MASS* - functions and datasets for "Modern Applied Statistics with S",
- *ggplot2* - grammar of graphics plots,
- *shiny* - interactive web graphics from R,
- *slidify* - HTML5 slide shows from R,
- *devtools* - create R packages,
- *roxygen2* - document R packages,
- *Rcpp* - integrate C++ code with R,
- *RcppArmadillo* - interface to Armadillo linear algebra library,
- *forecast* - linear models and forecasting,
- *tseries* - time series analysis and computational finance,
- *zoo* - time series and ordered objects,
- *xts* - advanced time series objects,
- *quantmod* - quantitative financial modeling framework,
- *caTools* - moving window statistics for graphics and time series objects,

CRAN Package Views

CRAN view for package *AER*:
http://cran.r-project.org/web/packages/AER/

Note:

- Authors,
- Version number,
- Reference manual,
- Vignettes,
- Dependencies on other packages.

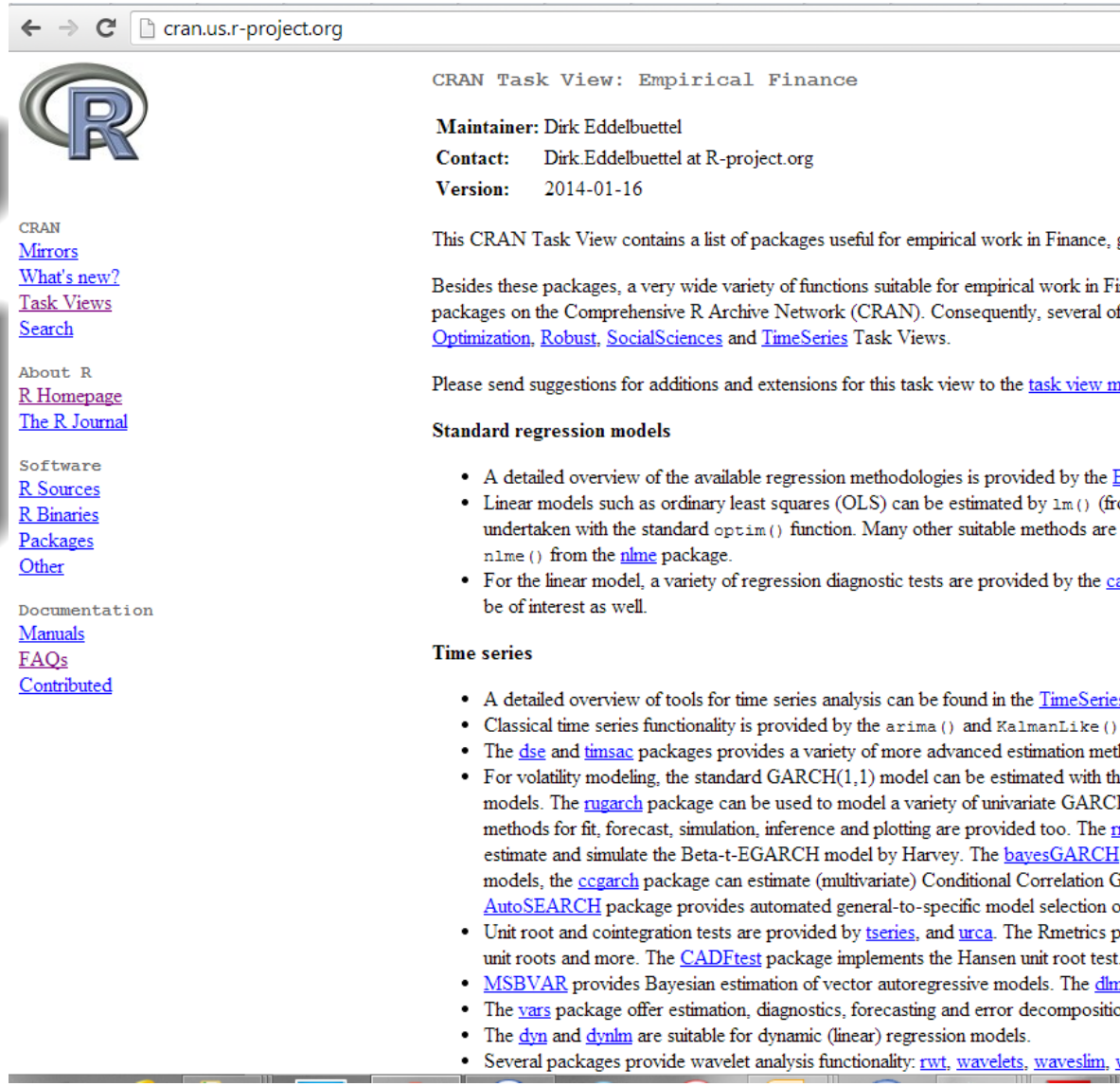
The package source code can be downloaded by clicking on the package source link,

CRAN Task Views

CRAN Finance Task View
http://cran.r-project.org//

Note:

- Maintainer,
- Topics,
- List of packages.



Installing Packages

Most packages need to be *installed* before they can be loaded and used.

Some packages like *MASS* are installed with base R (but not loaded).

Installing a package means downloading and saving its files to a local computer directory (hard disk), so they can be *loaded* by the R system.

The function `install.packages()` installs packages from the R command line.

Most widely used packages are available on the *CRAN* repository:

http://cran.r-project.org/web/packages/

Or on *R-Forge* or *GitHub*:

https://r-forge.r-project.org/

https://github.com/

Packages can also be installed in *RStudio* from the menu (go to Tools and then Install packages),

Packages residing on GitHub can be installed using the devtools packages.

```
> getOption("repos")  # get default package source
> .libPaths()  # get package save directory
> install.packages("AER")  # install "AER" from CRAN
> # install "PerformanceAnalytics" from R-Forge
> install.packages(
+   pkgs="PerformanceAnalytics",  # name
+   lib="C:/Users/Jerzy/Downloads",  # directory
+   repos="http://R-Forge.R-project.org")  # source
> # install devtools from CRAN
> install.packages("devtools")
> # load devtools
> library(devtools)
> # install package "babynamev" from GitHub
> install_github(repo="hadley/babynamev")
```

Installing Packages From Source

Sometimes packages aren't available in compiled form, so it's necessary to install them from their source code.

To install a package from source, the user needs to first install compilers and development tools:

For Windows install `Rtools`:

https://cran.r-project.org/bin/windows/Rtools/

For Mac OSX install `XCode` developer tools:

https://developer.apple.com/xcode/downloads/

The function `install.packages()` with argument `type="source"` installs a package from source.

The function `download.packages()` downloads the package's installation files (compressed tar format) to a local directory.

The function `install.packages()` can then be used to install the package from the downloaded files.

```
> # install package "PortfolioAnalytics" from source
> install.packages("PortfolioAnalytics",
+   type="source",
+   repos="http://r-forge.r-project.org")
> # download files for package "PortfolioAnalytics"
> download.packages(pkgs = "PortfolioAnalytics",
+   destdir = ".",  # download to cwd
+   type = "source",
+   repos="http://r-forge.r-project.org")
> # install "PortfolioAnalytics" from local tar source
> install.packages(
+   "C:/Users/Jerzy/Downloads/PortfolioAnalytics_0.9.3598.tar.gz",
+   repos=NULL, type="source")
```

Installed Packages

`defaultPackages` contains a list of packages loaded on startup by default.

The function `installed.packages()` returns a matrix of all packages installed on the system.

```
> getOption("defaultPackages")
> # matrix of installed package information
> packinfo <- installed.packages()
> dim(packinfo)
> # get all installed package names
> sort(unname(packinfo[, "Package"]))
> # get a few package names and their versions
> packinfo[sample(x=1:100, 5), c("Package", "Version")]
> # get info for package "xts"
> t(packinfo["xts", ])
```

Package Files and Directories

Package installation files are organized into multiple directories, including some of the following:

- ~/R containing R source code files,
- ~/src containing C++ and Fortran source code files,
- ~/data containing datasets,
- ~/man containing documentation files,

```
> # list directories in "PortfolioAnalytics" sub-directory
> gsub(
+   "C:/Users/Jerzy/Documents/R/win-library/3.1",
+   "~",
+   list.dirs(
+     file.path(
+       .libPaths()[1],
+       "PortfolioAnalytics")))
character(0)
```

Loading Packages

Most packages need to be *loaded* before they can be used in an R session.

Loading a package means attaching the package *namespace* to the *search path*, which allows R to call the package functions and data.

The functions `library()` and `require()` load packages, but in slightly different ways.

`library()` produces an *error* (halts execution) if the package can't be loaded.

`require()` returns `TRUE` if the package is loaded successfully, and `FALSE` otherwise.

Therefore `library()` is usually used in script files that might be sourced, while `require()` is used inside functions.

```
> # load package, produce error if can't be loaded
> library(MASS)
> # load package, return TRUE if loaded successfully
> require(MASS)
> # load quietly
> library(MASS, quietly=TRUE)
> # load without any messages
> suppressMessages(library(MASS))
> # remove package from search path
> detach(MASS)
> # install package if it can't be loaded successfully
> if (!require("xts")) install.packages("xts")
```

Referencing Package Objects

After a package is *loaded*, the package functions and data can be accessed by name.

Package objects can also be accessed without *loading* the package, by using the double-colon "::" reference operator.

For example, `TTR::VWAP()` references the function `VWAP()` from the package *TTR*.

This way users don't have to load the package *TTR* (with `library(TTR)`) to use functions from the package *TTR*.

Using the "::" operator displays the source of objects, and makes R code easier to analyze.

```
> # calculate VTI volume-weighted average price
> vwapv <- TTR::VWAP(
+   price=quantmod::Cl(rutils::etfenv$VTI),
+   volume=quantmod::Vo(rutils::etfenv$VTI), n=10)
```

Exploring Packages

The package *Ecdat* contains data sets for econometric analysis.

The data frame `Garch` contains daily currency prices.

The function `data()` loads external data or listv data sets in a package.

Some packages provide *lazy loading* of their data sets, which means they automatically load their data sets when they're needed (when they are called by some operation).

The package's data isn't loaded into R memory when the package is *loaded*, so it's not listed using `ls()`, but the package data is available without calling the function `data()`.

The function `data()` isn't required to load data sets that are set up for *lazy loading*.

```
> library()  # list all packages installed on the system
> search()  # list all loaded packages on search path
>
> # get documentation for package "Ecdat"
> packageDescription("Ecdat")  # get short description
> help(package="Ecdat")  # load help page
> library(Ecdat)  # load package "Ecdat"
> data(package="Ecdat")  # list all datasets in "Ecdat"
> ls("package:Ecdat")  # list all objects in "Ecdat"
> browseVignettes("Ecdat")  # view package vignette
> detach("package:Ecdat")  # remove Ecdat from search path
```

```
> library(Ecdat)  # load econometric data sets
> class(Garch)  # Garch is a data frame from "Ecdat"
> dim(Garch)  # daily currency prices
> head(Garch[, -2])  # col 'dm' is Deutsch Mark
> detach("package:Ecdat")  # remove Ecdat from search path
```

Package Namespaces

Package *namespaces*:

- Provide a mechanism for calling objects from a package,
- Hide functions and data internal to the package,
- Prevent naming conflicts between user and package names,

When a package is loaded using `library()` or `require()`, its *namespace* is attached to the search path.

```
> search()  # get search path for R objects
> library(MASS)  # load package "MASS"
> head(ls("package:MASS"))  # list some objects in "MASS"
> detach("package:MASS")  # remove "MASS" from search path
```

Package Namespaces and the Search Path

Packages may be loaded without their *namespace* being attached to the search path.

When packages are loaded, then packages they depend on are also loaded, but their *namespaces* aren't necessarily attached to the search path.

The function `loadedNamespaces()` lists all the loaded *namespaces*, including those that aren't on the search path.

The function `search()` returns the current search path for `R` objects.

`search()` returns many package *namespaces*, but not all the loaded *namespaces*.

```
> loadedNamespaces()  # get names of loaded namespaces
>
> search()  # get search path for R objects
```

Not Attached Namespaces

The function `sessionInfo()` returns information about the current R session, including packages that are loaded, but *not attached* to the search path.

`sessionInfo()` lists those packages as "loaded via a *namespace* (and not attached)"

```
> # get session info,
> # including packages not attached to the search path
> sessionInfo()
```

Non-Visible Objects

Non-visible objects (variables or functions) are either:

- objects from *not attached namespaces*,
- objects *not exported* outside a package,

Objects from packages that aren't attached can be accessed using the double-colon "::" reference operator.

Objects that are *not exported* outside a package can be accessed using the triple-colon ":::" reference operator.

Colon operators automatically load the associated package.

Non-visible objects in namespaces often use the ".*" name syntax.

```
> plot.xts  # package xts isn't loaded and attached
> head(xts::plot.xts, 3)
> methods("cbind")  # get all methods for function "cbind"
> stats::cbind.ts  # cbind isn't exported from package stats
> stats:::cbind.ts  # view the non-visible function
> getAnywhere("cbind.ts")
> library(MASS)  # load package 'MASS'
> select  # code of primitive function from package 'MASS'
```

Exploring Namespaces and Non-Visible Objects

The function `getAnywhere()` displays information about `R` objects, including non-visible objects.

Objects referenced *within* packages have different search paths than other objects:
Their search path starts in the package *namespace*, then the global environment and then finally the regular search path.

This way references to objects from `within` a package are resolved to the package, and they're not masked by objects of the same name in other environments.

```
> getAnywhere("cbind.ts")
```

Package *tseries* for Time Series Analysis

The package *tseries* contains functions for time series analysis and computational finance, such as:

- downloading historical data,
- plotting time series,
- calculating risk and performance measures,
- statistical *hypothesis testing*,
- calibrating models to time series,
- portfolio optimization,

Package *tseries* accepts time series of class `"ts"` and `"zoo"`, and also has its own class `"irts"` for irregular spaced time-series objects.

The package *zoo* is designed for managing *time series* and ordered data objects.

The function `zoo::coredata()` extracts the underlying numeric data from a complex data object.

```
> # Get documentation for package tseries
> packageDescription("tseries")  # Get short description
>
> help(package="tseries")  # Load help page
>
> library(tseries)  # Load package tseries
>
> data(package="tseries")  # List all datasets in "tseries"
>
> ls("package:tseries")  # List all objects in "tseries"
>
> detach("package:tseries")  # Remove tseries from search path
```

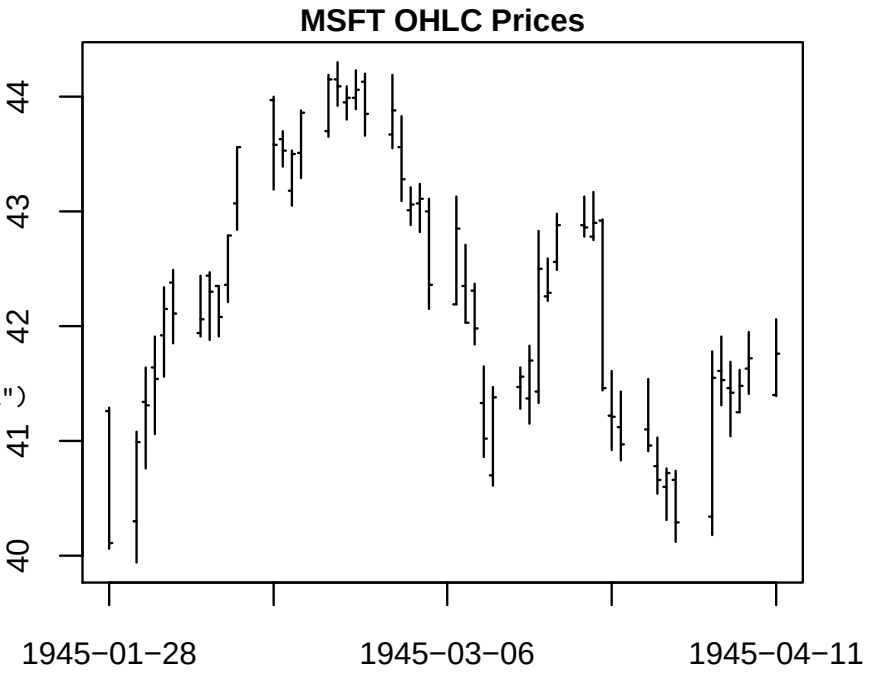
Plotting *OHLC* Time Series Using Package *tseries*

The package *tseries* contains functions for plotting time series:

- `seqplot.ts()` for plotting two time series in same panel.
- `plotOHLC()` for plotting *OHLC* time series.

The function `plotOHLC()` from package *tseries* plots *OHLC* time series.

```
> load(file="/Users/jerzy/Develop/lecture_slides/data/zoo_data.RData")
> # Get start and end dates
> datev <- time(stxts_adj)
> endd <- datev[NROW(datev)]
> startd <- round((4*endd + datev[1])/5)
> # Plot using plotOHLC
> plotOHLC(window(stxts_adj,
+           start=startd,
+           end=endd)[, 1:4],
+    xlab="", ylab="")
> title(main="MSFT OHLC Prices")
```



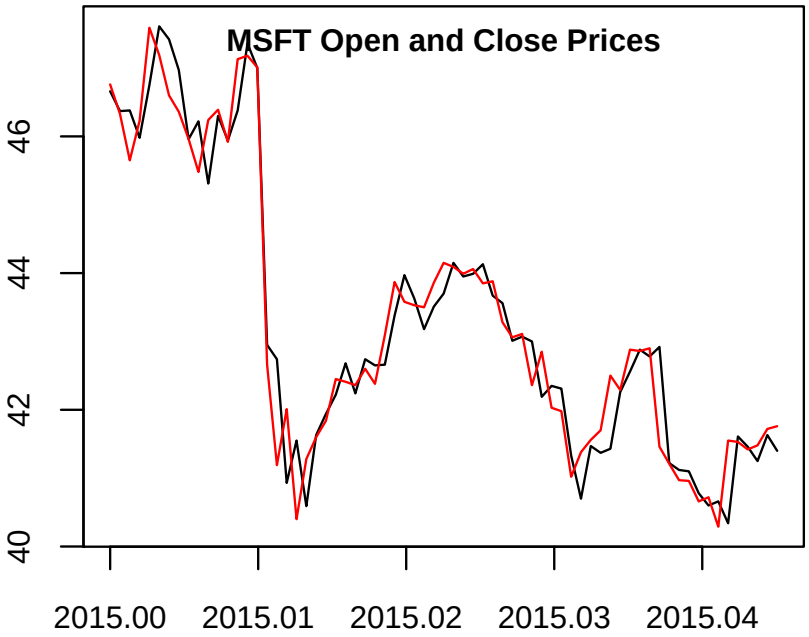
Plotting Two Time Series Using *tseries*

The function `seqplot.ts()` from package *tseries* plots two time series in same panel.

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

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

```
> library(lubridate)  # Load lubridate
> # Get start and end dates of msft
> startd <- lubridate::decimal_date(start(msft))
> endd <- lubridate::decimal_date(end(msft))
> # Calculate frequency of msft
> tstep <- NROW(msft)/(endd-startd)
> # Extract data from msft
> datav <- zoo::coredata(
+   window(msft, start=as.Date("2015-01-01"),
+    end=end(msft)))
> # Create ts object using ts()
> stxts <- ts(data=datav,
+   start=lubridate::decimal_date(as.Date("2015-01-01")),
+   frequency=tstep)
> seqplot.ts(x=stxts[, 1], y=stxts[, 4], xlab="", ylab="")
> title(main="MSFT Open and Close Prices", line=-1)
```



The function `zoo::coredata()` extracts the underlying numeric data from a complex data object.

Risk and Performance Estimation Using *tseries*

The package *tseries* contains functions for calculating risk and performance:

- `maxdrawdown()` for calculating the maximum drawdown,
- `sharpe()` for calculating the *Sharpe* ratio (defined as the excess return divided by the standard deviation),
- `sterling()` for calculating the *Sterling* ratio (defined as the return divided by the maximum drawdown),

```
> library(tseries)  # Load package tseries
> # Calculate maximum drawdown
> maxdrawdown(msft_adj[, "AdjClose"])
> max_drawd <- maxdrawdown(msft_adj[, "AdjClose"])
> zoo::index(msft_adj)[max_drawd$from]
> zoo::index(msft_adj)[max_drawd$to]
> # Calculate Sharpe ratio
> sharpe(msft_adj[, "AdjClose"])
> # Calculate Sterling ratio
> sterling(as.numeric(msft_adj[, "AdjClose"]))
```

Hypothesis Testing Using *tseries*

The package *tseries* contains functions for testing statistical hypothesis on time series:

- `jarque.bera.test()` *Jarque-Bera* test for normality of distribution of returns,
- `adf.test()` *Augmented Dickey-Fuller* test for existence of unit roots,
- `pp.test()` *Phillips-Perron* test for existence of unit roots,
- `kpss.test()` *KPSS* test for stationarity,
- `po.test()` *Phillips-Ouliaris* test for cointegration,
- `bds.test()` *BDS* test for randomness,

```
> msft <- suppressWarnings(  # Load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(msft)
> dim(msft)
> tail(msft, 4)
>
> # Calculate Sharpe ratio
> sharpe(msft[, "Close"], r=0.01)
> # Add title
> plot(msft[, "Close"], xlab="", ylab="")
> title(main="MSFT Close Prices", line=-1)
```

Calibrating Time Series Models Using *tseries*

The package *tseries* contains functions for calibrating models to time series:

- `garch()` for calibrating *GARCH* volatility models,
- `arma()` for calibrating *ARMA* models,

```
> library(tseries)  # Load package tseries
> msft <- suppressWarnings(  # Load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(msft)
> dim(msft)
> tail(msft, 4)
>
> # Calculate Sharpe ratio
> sharpe(msft[, "Close"], r=0.01)
> # Add title
> plot(msft[, "Close"], xlab="", ylab="")
> title(main="MSFT Close Prices", line=-1)
```

Portfolio Optimization Using *tseries*

The package *tseries* contains functions for miscellaneous functions:

`portfolio.optim()` for calculating mean-variance efficient portfolios.

```
> library(tseries)  # Load package tseries
> msft <- suppressWarnings(  # Load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(msft)
> dim(msft)
> tail(msft, 4)
>
> # Calculate Sharpe ratio
> sharpe(msft[, "Close"], r=0.01)
> # Add title
> plot(msft[, "Close"], xlab="", ylab="")
> title(main="MSFT Close Prices", line=-1)
```

Package *quantmod* for Quantitative Financial Modeling

Package *quantmod* for Quantitative Financial Modeling

The package *quantmod* is designed for downloading, manipulating, and visualizing *OHLC* time series data.

quantmod operates on time series of class `"xts"`, and provides many useful functions for building quantitative financial models:

- `getSymbols()` for downloading data from external sources (*Yahoo*, *FRED*, etc.),
- `getFinancials()` for downloading financial statements,
- `adjustOHLC()` for adjusting *OHLC* data,
- `Op()`, `Cl()`, `Vo()`, etc. for extracting *OHLC* data columns,
- `periodReturn()`, `dailyReturn()`, etc. for calculating periodic returns,
- `chartSeries()` for candlestick plots of *OHLC* data,
- `addBBands()`, `addMA()`, `addVo()`, etc. for adding technical indicators (Moving Averages, Bollinger Bands) and volume data to a plot,

```
> # Load package quantmod
> library(quantmod)
> # Get documentation for package quantmod
> # Get short description
> packageDescription("quantmod")
> # Load help page
> help(package="quantmod")
> # List all datasets in "quantmod"
> data(package="quantmod")
> # List all objects in "quantmod"
> ls("package:quantmod")
> # Remove quantmod from search path
> detach("package:quantmod")
```

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

The function `chartSeries()` from package *quantmod* can produce a variety of plots for *OHLC* time series, including candlestick plots, bar plots, and line plots.

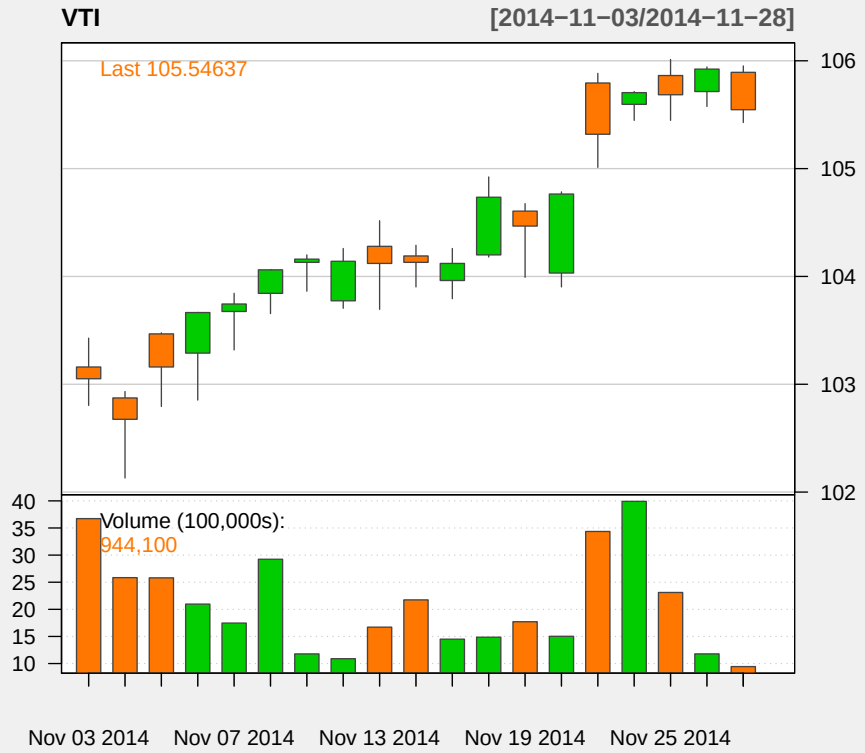
The argument `"type"` determines the type of plot (candlesticks, bars, or lines).

The argument `"theme"` accepts a `"chart.theme"` object, containing parameters that determine the plot appearance (colors, size, fonts).

`chartSeries()` automatically plots the volume data in a separate panel.

Candlestick plots are designed to visualize *OHLC* time series.

```
> # Plot OHLC candlechart with volume
> chartSeries(etfenv$VTI["2014-11"],
+       name="VTI",
+       theme=chartTheme("white"))
> # Plot OHLC bar chart with volume
> chartSeries(etfenv$VTI["2014-11"],
+       type="bars",
+       name="VTI",
+       theme=chartTheme("white"))
```



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.

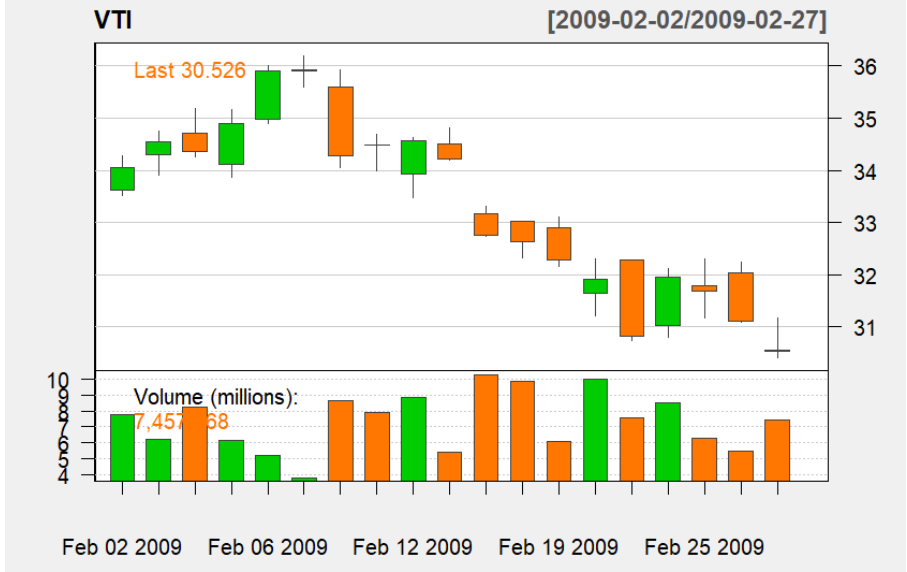
The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

Redrawing Plots Using `reChart()`

The function `reChart()` redraws plots using the same data set, but using additional parameters that control the plot appearance.

The argument `"subset"` allows subsetting the data to a smaller range of dates.

```
> # Plot OHLC candlechart with volume
> chartSeries(etfenv$VTI["2008-11/2009-04"], name="VTI")
> # Redraw plot only for Feb-2009, with white theme
> reChart(subset="2009-02", theme=chartTheme("white"))
```



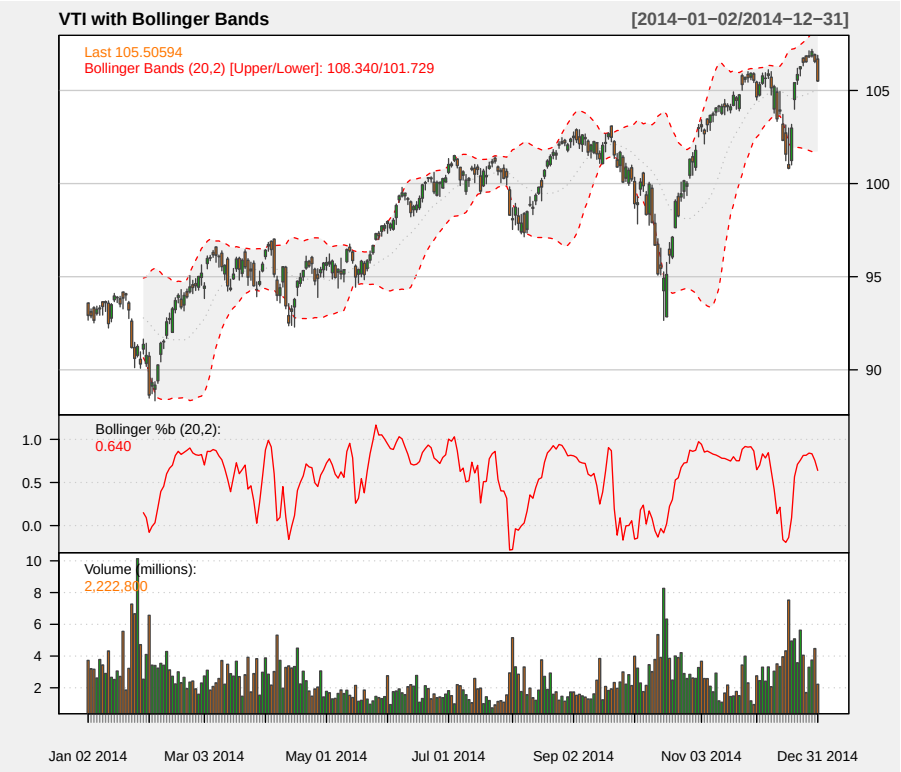
Plotting Technical Indicators Using `chartSeries()`

The argument `"TA"` allows adding technical indicators to the plot.

The technical indicators are functions provided by the package *TTR*.

The function `newTA()` allows defining new technical indicators.

```
> # Candlechart with Bollinger Bands
> chartSeries(etfenv$VTI["2014"],
+       TA="addBBands(): addBBands(draw='percent'): addVo()",
+       name="VTI with Bollinger Bands",
+       theme=chartTheme("white"))
> # Candlechart with two Moving Averages
> chartSeries(etfenv$VTI["2014"],
+       TA="addVo(): addEMA(10): addEMA(30)",
+       name="VTI with Moving Averages",
+       theme=chartTheme("white"))
> # Candlechart with Commodity Channel Index
> chartSeries(etfenv$VTI["2014"],
+       TA="addVo(): addBBands(): addCCI()",
+       name="VTI with Technical Indicators",
+       theme=chartTheme("white"))
```



Adding Indicators and Lines Using addTA()

The function addTA() adds indicators and lines to plots, and allows plotting lines representing a single vector of data.

The addTA() function argument "on" determines on which plot panel (subplot) the indicator is drawn.

"on=NA" is the default, and draws in a new plot panel below the existing plot.

"on=1" draws in the foreground of the main plot panel, and "on=-1" draws in the background.

```
> ohlc <- rutils::etfenv$VTI["2009-02/2009-03"]
> VTI_close <- quantmod::Cl(ohlc)
> VTI_vol <- quantmod::Vo(ohlc)
> # Calculate volume-weighted average price
> vwapv <- TTR::VWAP(price=VTI_close, volume=VTI_vol, n=10)
> # Plot OHLC candlechart with volume
> chartSeries(ohlc, name="VTI plus VWAP", theme=chartTheme("white"))
> # Add VWAP to main plot
> addTA(ta=vwapv, on=1, col='red')
> # Add price minus VWAP in extra panel
> addTA(ta=(VTI_close-vwapv), col='red')
```

The function VWAP() from package *TTR* calculates the Volume Weighted Average Price as the average of past prices multiplied by their trading volumes, divided by the total volume.

The argument "n" represents the number of look-back intervals used for averaging,

Shading Plots Using addTA()

addTA() accepts Boolean vectors for shading of plots.

The function addLines() draws vertical or horizontal lines in plots.

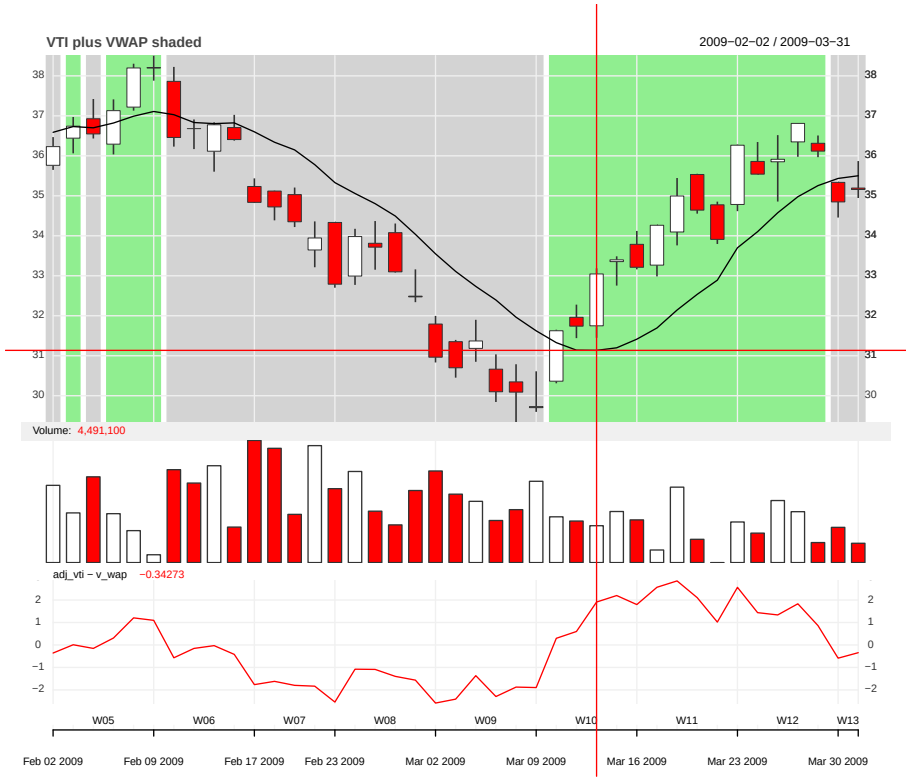
```
> # Plot OHLC candlechart with volume
> chartSeries(ohlc, name="VTI plus VWAP shaded",
+       theme=chartTheme("white"))
> # Add VWAP to main plot
> addTA(ta=vwapv, on=1, col='red')
> # Add price minus VWAP in extra panel
> addTA(ta=(VTI_close-vwapv), col='red')
> # Add background shading of areas
> addTA((VTI_close-vwapv) > 0, on=-1,
+ col="lightgreen", border="lightgreen")
> addTA((VTI_adj-vwapv) < 0, on=-1,
+ col="lightgrey", border="lightgrey")
> # Add vertical and horizontal lines at vwapv minimum
> addLines(v=which.min(vwapv), col='red')
> addLines(h=min(vwapv), col='red')
```

Plotting Time Series Using chart_Series()

The function chart_Series() from package *quantmod* is an improved version of chartSeries(), with better aesthetics.

chart_Series() plots are compatible with the base graphics package in R, so that standard plotting functions can be used in conjunction with chart_Series().

```
> # OHLC candlechart VWAP in main plot,
> chart_Series(x=ohlc, # Volume in extra panel
+         TA="add_Vo(); add_TA(vwapv, on=1)",
+         name="VTI plus VWAP shaded")
> # Add price minus VWAP in extra panel
> add_TA(VTI_adj-vwapv, col='red')
> # Add background shading of areas
> add_TA((VTI_adj-vwapv) > 0, on=-1,
+ col="lightgreen", border="lightgreen")
> add_TA((VTI_adj-vwapv) < 0, on=-1,
+ col="lightgrey", border="lightgrey")
> # Add vertical and horizontal lines
> abline(v=which.min(vwapv), col='red')
> abline(h=min(vwapv), col='red')
```



chart_Series() also has its own functions for adding indicators: add_TA(), add_BBands(), etc.

Note that functions associated with chart_Series() contain an underscore in their name,

Plot and Theme Objects of chart_Series()

The function chart_Series() creates a *plot object* and returns it *invisibly*.

A *plot object* is an environment of class *replot*, containing parameters specifying a plot.

A plot can be rendered by calling, plotting, or printing the *plot object*.

A plot *theme object* is a list containing parameters that determine the plot appearance (colors, size, fonts).

The function chart_theme() returns the *theme object*.

chart_Series() plots can be modified by modifying *plot objects* or *theme objects*.

Plot and theme objects can be modified directly, or by using accessor and setter functions.

The parameter "plot=FALSE" suppresses plotting and allows modifying *plot objects*.

```
> # Extract plot object
> chobj <- chart_Series(x=ohlc, plot=FALSE)
> class(chobj)
> ls(chobj)
> class(chobj$get_ylim)
> class(chobj$set_ylim)
> # ls(chobj$Env)
> class(chobj$Env$actions)
> plotheme <- chart_theme()
> class(plotheme)
> ls(plotheme)
```

Customizing chart_Series() Plots

chart_Series() plots can be customized by modifying the plot and theme objects.

Plot and theme objects can be modified directly, or by using accessor and setter functions.

A plot is rendered by calling, plotting, or printing the plot object.

The parameter "plot=FALSE" suppresses plotting and allows modifying *plot objects*.

```
> ohlc <- rutils::etfenv$VTI["2010-04/2010-05"]
> # Extract, modify theme, format tick marks "%b %d"
> plotheme <- chart_theme()
> plotheme$format.labels <- "%b %d"
> # Create plot object
> chobj <- chart_Series(x=ohlc, theme=plotheme, plot=FALSE)
> # Extract ylim using accessor function
> ylim <- chobj$get_ylim()
> ylim[[2]] <- structure(range(quantmod::Cl(ohlc)) + c(-1, 1),
+   fixed=TRUE)
> # Modify plot object to reduce y-axis range
> chobj$set_ylim(ylim)  # use setter function
> # Render the plot
> plot(chobj)
```

Plotting chart_Series() in Multiple Panels

chart_Series() plots are compatible with the base graphics package, allowing easy plotting in multiple panels.

The parameter "plot=FALSE" suppresses plotting and allows adding extra plot elements.

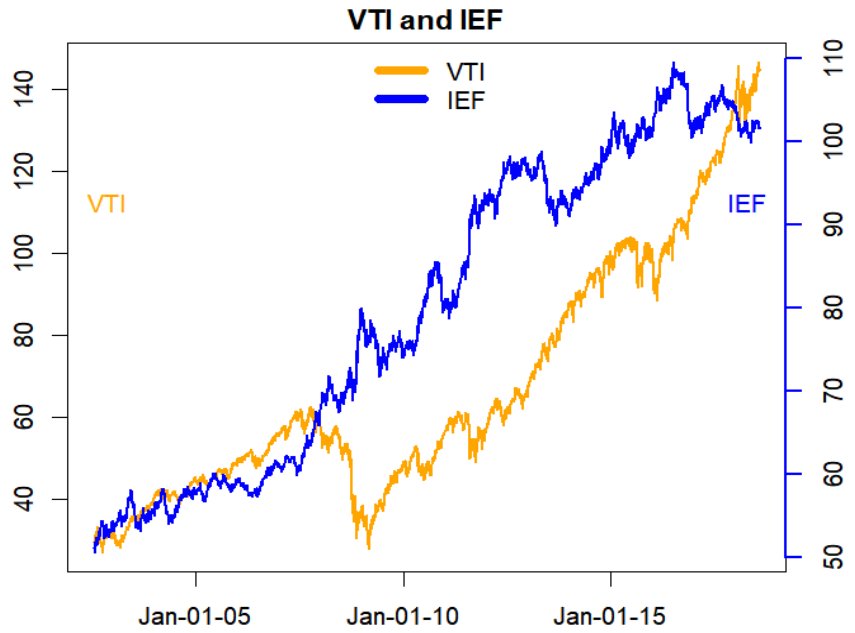
```
> # Calculate VTI and XLF volume-weighted average price
> vwapv <- TTR::VWAP(price=quantmod::Cl(rutils::etfenv$VTI),
+       volume=quantmod::Vo(rutils::etfenv$VTI), n=10)
> XLF_vwap <- TTR::VWAP(price=quantmod::Cl(rutils::etfenv$XLF),
+       volume=quantmod::Vo(rutils::etfenv$XLF), n=10)
> # Open graphics device, and define
> # Plot area with two horizontal panels
> x11(); par(mfrow=c(2, 1))
> chobj <- chart_Series(  # Plot in top panel
+   x=etfenv$VTI["2009-02/2009-04"],
+   name="VTI", plot=FALSE)
> add_TA(vwapv["2009-02/2009-04"], lwd=2, on=1, col='blue')
> # Plot in bottom panel
> chobj <- chart_Series(x=etfenv$XLF["2009-02/2009-04"],
+   name="XLF", plot=FALSE)
> add_TA(XLF_vwap["2009-02/2009-04"], lwd=2, on=1, col='blue')
```

zoo Plots With Two "y" Axes

The function `plot.zoo()` plots time series.

The function `axis()` plots customized axes in an existing plot.

```
> # Open plot window and set plot margins
> x11(width=6, height=4)
> par(mar=c(2, 2, 2, 2), oma=c(1, 1, 1, 1))
> # Plot first time series without x-axis
> zoo::plot.zoo(pricev[, 1], lwd=2, col="orange",
+         xlab=NA, ylab=NA, xaxt="n")
> # Create X-axis date labels and add X-axis
> datev <- pretty(zoo::index(pricev))
> axis(side=1, at=datev, labels=format(datev, "%b-%d-%y"))
> # Plot second time series without y-axis
> par(new=TRUE)  # Allow new line on same plot
> zoo::plot.zoo(pricev[, 2], xlab=NA, ylab=NA,
+         lwd=2, yaxt="n", col="blue", xaxt="n")
> # Plot second y-axis on right
> axis(side=4, lwd=2, col="blue")
> # Add axis labels
> mtext(colv[1], cex=1.2, lwd=3, side=2, las=2, adj=(-0.5), padj=(-5), col="orange")
> mtext(colv[2], cex=1.2, lwd=3, side=4, las=2, adj=1.5, padj=(-5), col="blue")
> # Add title and legend
> title(main=paste(colv, collapse=" and "), line=0.5)
> legend("top", legend=colv, cex=1.0, bg="white",
+  lty=1, lwd=6, col=c("orange", "blue"), bty="n")
```



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)
> # Calculate volume-weighted average price
> ohlc <- rutils::etfenv$VTI
> vwapv <- TTR::VWAP(price=quantmod::Cl(ohlc),
+     volume=quantmod::Vo(ohlc), n=20)
> # Add VWAP to OHLC data
> datav <- cbind(ohlc[, 1:4], vwapv)["2009-01/2009-04"]
> # Create dygraphs object
> dyplot <- dygraphs::dygraph(datav)
> # Increase line width and color
> dyplot <- dygraphs::dyOptions(dyplot,
+   colors="red", strokeWidth=3)
> # Convert dygraphs object to candlestick plot
> dyplot <- dygraphs::dyCandlestick(dyplot)
> # Render candlestick plot
> dyplot
> # Candlestick plot using pipes syntax
> dygraphs::dygraph(datav) %>% dyCandlestick() %>%
+   dyOptions(colors="red", strokeWidth=3)
> # Candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dyOptions(dygraphs::dygraph(datav),
+   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*,

dygraphs OHLC Plots With Background Shading

The function `dyShading()` adds shading to a *dygraphs* plot object.

The function `dyShading()` requires a vector of dates for shading.

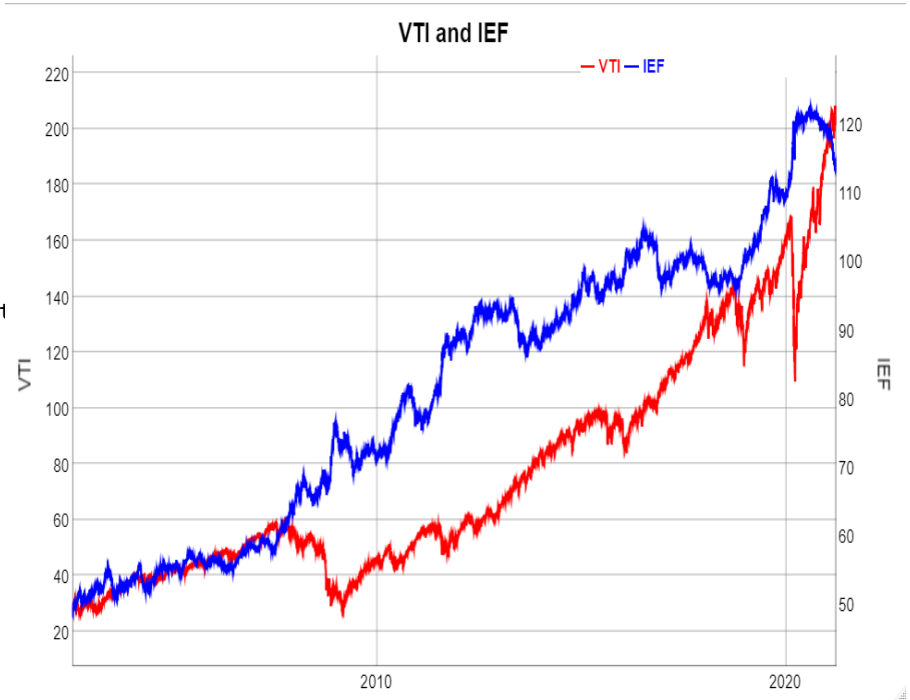
```
> # Create candlestick plot with background shading
> indic <- (quantmod::Cl(datav) > datav[, "VWAP"])
> whichv <- which(rutils::diffit(indic) != 0)
> indic <- rbind(first(indic), indic[whichv, ], last(indic))
> datev <- zoo::index(indic)
> indic <- ifelse(drop(coredata(indic)), "lightgreen", "antiquewhite
> # Create dygraph object without rendering it
> dyplot <- dygraphs::dygraph(datav) %>% dyCandlestick() %>%
+   dyOptions(colors="red", strokeWidth=3)
> # Add shading
> for (i in 1:(NROW(indic)-1)) {
+     dyplot <- dyplot %>%
+ dyShading(from=datev[i], to=datev[i+1], color=indic[i])
+ }  # end for
> # Render the dygraph object
> dyplot
```

dygraphs Plots With Two "y" Axes

The function `dyAxis()` from package *dygraphs* plots customized axes to a *dygraphs* plot object.

The function `dySeries()` adds a time series to a *dygraphs* plot object.

```
> library(dygraphs)
> # Prepare VTI and IEF prices
> pricev <- cbind(quantmod::Cl(rutils::etfenv$VTI), quantmod::Cl(rut
> pricev <- na.omit(pricev)
> colv <- rutils::get_name(colnames(pricev))
> colnames(pricev) <- colv
> # dygraphs plot with two y-axes
> library(dygraphs)
> dygraphs::dygraph(pricev, main=paste(colv, collapse=" and ")) %>%
+   dyAxis(name="y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis(name="y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", strokeWidth=2, col="red") %>%
+   dySeries(name=colv[2], axis="y2", strokeWidth=2, col="blue")
```



draft: Package *qmao* for Quantitative Financial Modeling

The package *qmao* is designed for downloading, manipulating, and visualizing *OHLC* time series data, package *quantmod*

qmao uses time series of class `"xts"`, and provides many useful functions for building quantitative financial models:

- `getSymbols()` for downloading data from external sources (*Yahoo*, *FRED*, etc.),
- `getFinancials()` for downloading financial statements,
- `adjustOHLC()` for adjusting *OHLC* data,
- `Op()`, `Cl()`, `Vo()`, etc. for extracting *OHLC* data columns,
- `periodReturn()`, `dailyReturn()`, etc. for calculating periodic returns,
- `chartSeries()` for candlestick plots of *OHLC* data,
- `addBBands()`, `addMA()`, `addVo()`, etc. for adding technical indicators (Moving Averages, Bollinger Bands) and volume data to a plot,

```
> # Load package qmao
> library(qmao)
> # Get documentation for package qmao
> # Get short description
> packageDescription("qmao")
> # Load help page
> help(package="qmao")
> # List all datasets in "qmao"
> data(package="qmao")
> # List all objects in "qmao"
> ls("package:qmao")
> # Remove qmao from search path
> detach("package:qmao")
```