FRE7241 Algorithmic Portfolio Management

Lecture#1, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

January 23, 2018

NYU | TANDON SCHOOL OF ENGINEERING

FRE7241 Course Description and Objectives

Course Description

The course will apply the R programming language to *momentum trading*, *statistical arbitrage* (pairs trading), and other active portfolio management strategies. The course will implement volatility and price *forecasting models*, asset pricing and *factor models*, and *portfolio optimization*. The course will apply *machine learning* techniques, such as *backtesting* (cross-validation) and *parameter regularization* (shrinkage).

FRE7241 Course Description and Objectives

Course Description

The course will apply the `R` programming language to *momentum trading*, *statistical arbitrage* (pairs trading), and other active portfolio management strategies. The course will implement volatility and price *forecasting models*, asset pricing and *factor models*, and *portfolio optimization*. The course will apply *machine learning* techniques, such as *backtesting* (cross-validation) and *parameter regularization* (shrinkage).

Course Objectives

Students will learn through `R` coding exercises how to:

- download data from external sources, and to scrub and format it,
- estimate time series parameters, and fit models such as `ARIMA`, `GARCH`, and factor models,
- optimize portfolios under different constraints and risk-return objectives,
- backtest active portfolio management strategies and evaluate their performance,

FRE7241 Course Description and Objectives

Course Description

The course will apply the `R` programming language to *momentum trading*, *statistical arbitrage* (pairs trading), and other active portfolio management strategies. The course will implement volatility and price *forecasting models*, asset pricing and *factor models*, and *portfolio optimization*. The course will apply *machine learning* techniques, such as *backtesting* (cross-validation) and *parameter regularization* (shrinkage).

Course Objectives

Students will learn through `R` coding exercises how to:

- download data from external sources, and to scrub and format it,
- estimate time series parameters, and fit models such as `ARIMA`, `GARCH`, and factor models,
- optimize portfolios under different constraints and risk-return objectives,
- backtest active portfolio management strategies and evaluate their performance,

Course Prerequisites

FRE6123 Financial Risk Management and Asset Pricing. The `R` language is considered to be challenging, so this course requires some programming experience with other languages such as `C++` or `Python`. Students with less programming experience are encouraged to first take *FRE6871 R in Finance*. Students should also have knowledge of basic statistics (random variables, estimators, hypothesis testing, regression, etc.)

Homeworks and Tests

Homeworks and Tests

Grading will be based on homeworks and in-class tests. There will be no final exam.

The tests will require writing code, which should run directly when pasted into an `R` session, and should produce the required output, without any modifications.

Students will be allowed to consult course slides, and to copy code from them, and to copy from books or any online sources, but they will be required to provide references to those external sources (such as links or titles and page numbers).

The tests will be closely based on code contained in the course slides, so students are encouraged to become very familiar with those slides.

Students will submit their homework and test files only through *NYU Classes* (not emails).

Students will be required to bring their laptop computers to class and run the `R` Interpreter, and the `RStudio` Integrated Development Environment (*IDE*), during the lecture.

Homeworks will also include reading assignments designed to help prepare for tests.

Homeworks and Tests

Homeworks and Tests

Grading will be based on homeworks and in-class tests. There will be no final exam.

The tests will require writing code, which should run directly when pasted into an `R` session, and should produce the required output, without any modifications.

Students will be allowed to consult course slides, and to copy code from them, and to copy from books or any online sources, but they will be required to provide references to those external sources (such as links or titles and page numbers).

The tests will be closely based on code contained in the course slides, so students are encouraged to become very familiar with those slides.

Students will submit their homework and test files only through *NYU Classes* (not emails).

Students will be required to bring their laptop computers to class and run the `R` Interpreter, and the `RStudio` Integrated Development Environment (*IDE*), during the lecture.

Homeworks will also include reading assignments designed to help prepare for tests.

Graduate Assistant

The graduate assistant (GA) will be Zhenyu (Lucy) Yin zy944@nyu.edu.
The GA will answer questions during office hours, or via *NYU Classes* forums, not via emails.
Please send emails regarding lecture matters from *NYU Classes* (not personal emails).

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

The tests will require mostly copying code samples from the course slides, making some modifications to them, and combining them with other code samples,

Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer,

So don't leave test assignments unanswered, and instead copy any code samples from the course slides that are related to the solution and make sense,

Contact the GA during office hours via text or phone, and submit questions to the GA or to me via *NYU Classes*,

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

The tests will require mostly copying code samples from the course slides, making some modifications to them, and combining them with other code samples,

Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer,

So don't leave test assignments unanswered, and instead copy any code samples from the course slides that are related to the solution and make sense,

Contact the GA during office hours via text or phone, and submit questions to the GA or to me via *NYU Classes*,

Please Submit *Minimal Working Examples* With Your Questions

When submitting questions, students should provide a *minimal working example* that produces the error in `R`, with the following items:

- The *complete* `R` code that produces the error, including the seed value for random numbers,
- The version of `R` (output of command: `sessionInfo()`), and the versions of `R` packages,
- The type and version of your operating system (Windows or OSX),
- The dataset file used by the `R` code,
- The text or screenshots of error messages,

Course Grading Policies

Numerical Scores

Tests will be graded and assigned numerical scores. Each part of the tests will be graded separately and assigned a numerical score.
Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.
Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

Course Grading Policies

Numerical Scores

Tests will be graded and assigned numerical scores. Each part of the tests will be graded separately and assigned a numerical score.
Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.
Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

Plagiarism

Plagiarism (copying from other students) and cheating will be punished.
But copying code from course slides, books, or any online sources is allowed and encouraged.
Students must provide references to any external sources from which they copy code (such as links or titles and page numbers).

Course Grading Policies

Numerical Scores

Tests will be graded and assigned numerical scores. Each part of the tests will be graded separately and assigned a numerical score.
Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.
Partial credit will be given even for code that doesn't produce the correct output, but that has elements of code that can be useful for producing the right answer.

Plagiarism

Plagiarism (copying from other students) and cheating will be punished.
But copying code from course slides, books, or any online sources is allowed and encouraged.
Students must provide references to any external sources from which they copy code (such as links or titles and page numbers).

Letter Grades

Letter grades for the course will be derived from the cumulative scores obtained for all the tests.
Very high numerical scores close to the maximum won't guarantee an A letter grade, since grading will also depend on the difficulty of the assignments.

FRE7241 Course Materials

Course Slides

The course will be mostly self-contained, using detailed course slides containing extensive, working R code examples.
The course will also utilize data and tutorials which are freely available on the internet.

FRE7241 Course Materials

Course Slides

The course will be mostly self-contained, using detailed course slides containing extensive, working R code examples.
The course will also utilize data and tutorials which are freely available on the internet.

FRE7241 Recommended Textbooks

- *"Financial Risk Modelling"* introduces volatility models, portfolio optimization, and tactical asset allocation, with great review of R packages and examples in R:
  Bernhard Pfaff. *Financial Risk Modelling and Portfolio Optimization with R*. First Edition. Wiley, 2013. URL: http://www.pfaffikus.de/wiley.html
- *"Analysis for Financial Engineering"* introduces regression, cointegration, multivariate time series analysis, ARIMA, GARCH, CAPM, and factor models, with examples in R:
  David Ruppert. *Statistics and Data Analysis for Financial Engineering*. Second Edition. Springer Texts in Statistics. Springer, 2011. URL: http://legacy.orie.cornell.edu/davidr/SDAFE/
- *"Applied Econometrics"* introduces advanced statistical models and econometrics:
  Christian Kleiber and Achim Zeileis. *Applied Econometrics with R (Use R!)*. Springer, 2008. URL: http://eeecon.uibk.ac.at/~zeileis/teaching/AER/index.html
- *"Econometrics in R"* contains a good review of regression and time series analysis:
  http://cran.r-project.org/doc/contrib/Farnsworth-EconometricsInR.pdf

FRE7241 Supplementary Books

- *"The Art of R Programming"* good introduction to R and to statistical models:
  Norman Matloff. *The Art of R Programming*. First Edition. No Starch Press, 2011. URL: http://it-ebooks.info/book/1734/
- The *"Statistical Learning"* book introduces machine learning techniques using R - it's a must have for advanced finance applications (you can download it for free):
  Trevor Hastie Robert Tibshirani Gareth James Daniela Witten. *An Introduction to Statistical Learning with Applications in R*. First Edition. Springer, 2013. URL: http://www-bcf.usc.edu/~gareth/ISL/index.html
- *"Advanced R"* is the best book for learning the advanced features of R:
  Hadley Wickham. *Advanced R*. First Edition. Chapman and Hall/CRC, 2014. URL: http://adv-r.had.co.nz/
- *"Systematic Trading"* is a great introduction to the practice of systematic trading:
  Robert Carver. *Systematic Trading*. First Edition. Harriman House, 2015. URL: http://www.systematictrading.org/
  Robert Carver blog: http://qoppac.blogspot.com/
- *"Numerical Recipes"* is a great reference for linear algebra and numerical methods, implemented in working C++ code:
  William T. Vetterling Brian P. Flannery William H. Press Saul A. Teukolsky. *Numerical Recipes in C++*. Third Edition. Cambridge University Press, 2007. URL: http://www.nr.com/
- *"R in Action"* good introduction to R and to statistical models:

FRE7241 Supplementary Materials

`Notepad++` is a free source code editor for `MS Windows`, that supports several programming languages, including `R`.

`Notepad++` has a very convenient and fast *search and replace* function, that allows *search and replace* in multiple files.

http://notepad-plus-plus.org/

FRE7241 Supplementary Materials

`Notepad++` is a free source code editor for `MS Windows`, that supports several programming languages, including `R`.

`Notepad++` has a very convenient and fast *search and replace* function, that allows *search and replace* in multiple files.

http://notepad-plus-plus.org/

Introduction to Computational Finance with R

Good course by prof. Eric Zivot, with lots of `R` examples:
https://www.datacamp.com/courses/computational-finance-and-financial-econometrics-with-r/

Internal R Help and Documentation

The function `help()` displays documentation on a function or subject,

Preceding the keyword with a single `"?"` is equivalent to calling `help()`,

```
> # display documentation on function "getwd"
> help(getwd)
> ?getwd  # equivalent to "help(getwd)"
```

The function `help.start()` displays a page with links to internal documentation,

R documentation is also available in RGui under the help tab,

The *pdf* files with R documentation are also available directly under:
C:/Program Files/R/R-3.1.2/doc/manual/
(the exact path will depend on the R version.)

```
> help.start()  # open the hypertext documentati
```

"Introduction to R" by Venables and R Core Team:
Venables. *An Introduction to R*. URL: http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

R Online Help and Documentation

R Programming Wikibook

Wikibooks are crowdsourced textbooks
http://en.wikibooks.org/wiki/R_Programming/

R FAQ

Frequently Asked Questions about R
http://cran.r-project.org/doc/FAQ/R-FAQ.html

R-seek Online Search Tool

R-seek allows online searches specific to the R language
http://www.rseek.org/

R-help Mailing List

R-help is a very comprehensive Q&A mailing list
https://stat.ethz.ch/mailman/listinfo/r-help
R-help has archives of past Q&A - search it before you ask
https://stat.ethz.ch/pipermail/r-help/
GMANE allows searching the R-help archives using a usenet newsgroup style GUI
http://news.gmane.org/gmane.comp.lang.r.general

R Style Guides

DataCamp R style guide

The DataCamp R style guide is very close to what I have adopted:
DataCamp R style guide

Google R style guide

The Google R style guide is similar to DataCamp's:
Google R style guide

Stack Exchange

Stack Overflow

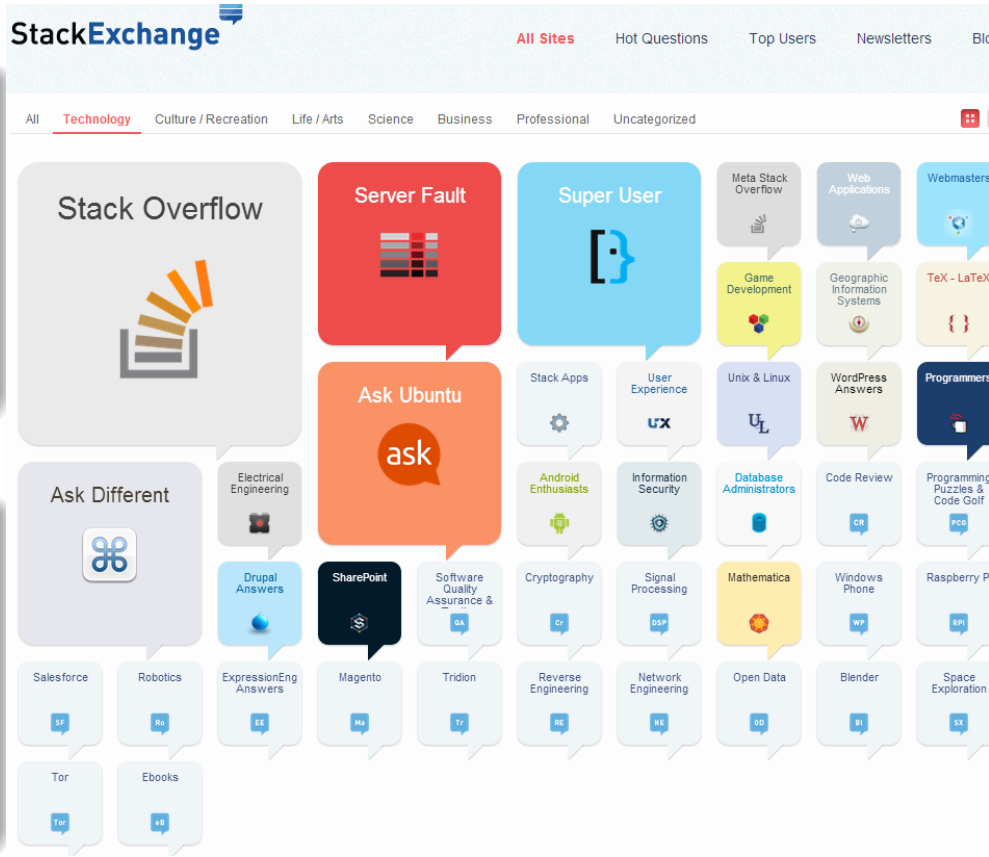
Stack Overflow is a Q&A forum for computer programming, and is part of Stack Exchange

- http://stackoverflow.com
- http://stackoverflow.com/questions/tagged/r
- http://stackoverflow.com/tags/r/info

Stack Exchange

Stack Exchange is a family of Q&A forums in a variety of fields

- http://stackexchange.com/
- http://stackexchange.com/sites#technology
- http://quant.stackexchange.com/



RStudio Support

RStudio has extensive online help, Q&A database, and documentation

https://support.rstudio.com/hc/en-us

https://support.rstudio.com/hc/en-us/sections/200107586-Using-RStudio

https://support.rstudio.com/hc/en-us/sections/200148796-Advanced-Topics

R Online Books and Courses

Companion website to the book *"Advanced R"* by Hadley Wickham - chief scientist at *RStudio*

The best book for learning the advanced features of R: http://adv-r.had.co.nz/

Endmemo web book

Good, but not interactive: http://www.endmemo.com/program/R/

Quick-R by Robert Kabacoff

Good, but not interactive: http://www.statmethods.net/

R for Beginners by Emmanuel Paradis

Good, basic introduction to R: http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

Cookbook for R by Winston Chang from *RStudio*

Good plotting, but not interactive: http://www.cookbook-r.com/

R Online Interactive Courses

Datacamp Interactive Courses

Datacamp introduction to R: https://www.datacamp.com/courses/introduction-to-r/

Datacamp list of free courses: https://www.datacamp.com/community/open-courses

Datacamp basic statistics in R: https://www.datacamp.com/community/open-courses/basic-statistics

Datacamp computational finance in R: https://www.datacamp.com/community/open-courses/computational-finance-and-financial-econometrics-with-r

Datacamp machine learning in R: https://www.datacamp.com/community/open-courses/kaggle-r-tutorial-on-machine-learning

Try R

Interactive R tutorial, but rather basic: http://tryr.codeschool.com/

R Blogs and Experts

R-Bloggers

R-Bloggers is an aggregator of blogs dedicated to R

http://www.r-bloggers.com/

Tal Galili is the author of R-Bloggers and has his own excellent blog

http://www.r-statistics.com/

Dirk Eddelbuettel

Dirk is a *Top Answerer* for R questions on Stackoverflow, the author of the Rcpp package, and the CRAN Finance View

http://dirk.eddelbuettel.com/

http://dirk.eddelbuettel.com/code/

http://dirk.eddelbuettel.com/blog/

http://www.rinfinance.com/

Romain Frangois

Romain is an R Enthusiast and Rcpp Hero

http://romainfrancois.blog.free.fr/

http://romainfrancois.blog.free.fr/index.php?tag/graphgallery

http://blog.r-enthusiasts.com/

More R Blogs and Experts

Revolution Analytics Blog

R blog by Revolution Analytics software vendor
http://blog.revolutionanalytics.com/

RStudio Blog

R blog by *RStudio*
http://blog.rstudio.org/

GitHub for Hosting Software Projects Online

GitHub is an internet-based online service for hosting repositories of software projects,

GitHub provides version control using *git* (designed by Linus Torvalds),

Most R projects are now hosted on *GitHub*,

Google uses *GitHub* to host its *tensorflow* library for machine learning:

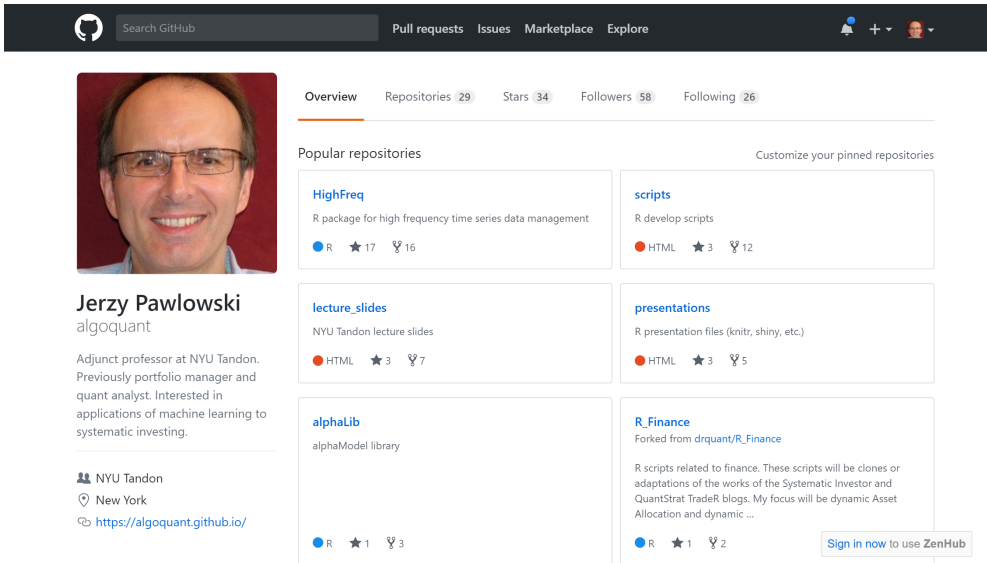
https://github.com/tensorflow/tensorflow

All the *FRE-7241* and *FRE-6871* lectures are hosted on *GitHub*:

https://github.com/algoquant/lecture_slides

https://github.com/algoquant

Hosting projects on *Google* is a great way to advertize your skills and network with experts,



What is R?

- Is an open-source software environment for statistical computing and graphics,
- Is an interpreted language, allowing interactive code development,
- Is a functional language where every operator is an R function,
- Supports object-oriented programming with *classes* and *methods*,
- Is a very expressive language that allows performing complex operations with very few lines of code,
- Has metaprogramming facilities that allow programming on the language,
- Is written in R itself and in C/C++,
- Has vectorized functions written in C/C++, allowing very fast execution of loops over vector elements,
- Is extended through user-created *packages* (function libraries), providing for the latest developments, such as *Machine Learning*,

http://www.r-project.org/
http://en.wikipedia.org/wiki/R_(programming_language)

Why is R More Difficult Than Other Languages?

R is more difficult than other languages because:

- R is a *functional* language, and the functional syntax may be unfamiliar to users of procedural languages like C/C++,
- There are many dozens of user-created *packages* (function libraries), and it's often difficult to tell which are the most useful or best for a particular application,
- The R interpreter produces very cryptic *warnings* and *error* messages,
- This is because the R interpreter is a programming environment, which means that it performs many additional complex calculations quietly (*under-the-hood*) to assist the user,
- But if there's a bug in the code or bad data then the complex functions underlying the R interpreter produce *warnings* and *error* messages, which exposeses the user to the complexity of the R interpreter,

This course is designed to teach the most useful elements of R for financial analysis, through case studies and examples,

The R License

R is open-source software released under the GNU General Public License:

http://www.r-project.org/Licenses

Some other R packages are released under the Creative Commons Attribution-ShareAlike License:

http://creativecommons.org

Installing R and *RStudio*

Students will be required to bring their laptop computers to all the lectures, and to run the R Interpreter and **RStudio** RStudio during the lecture,

Laptop computers will be necessary for following the lectures, and for performing tests,

Students will be required to install and to become proficient with the R Interpreter,
Students can download the R Interpreter from CRAN (Comprehensive R Archive Network):

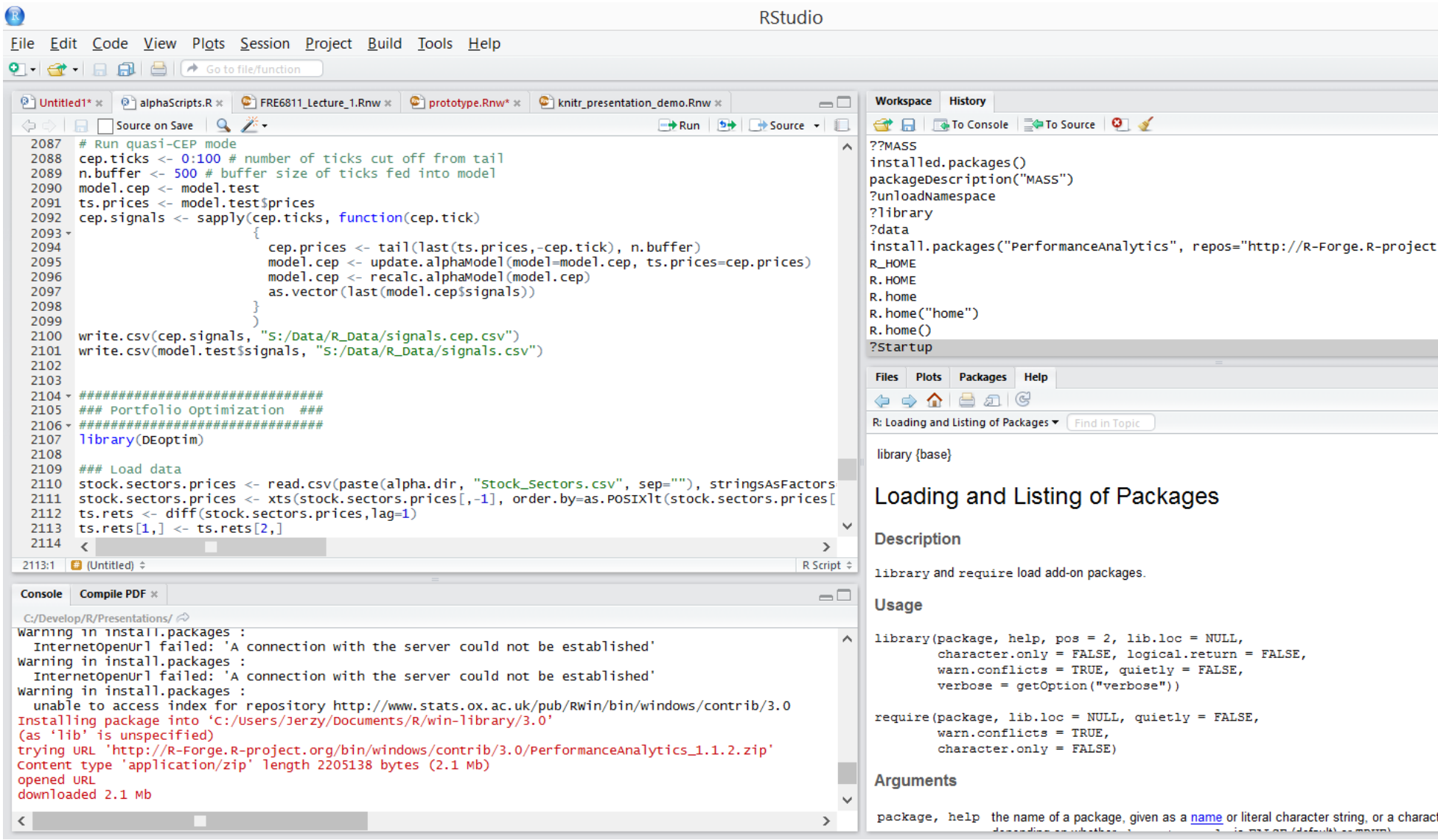
http://cran.r-project.org/

To invoke the RGui interface, click on:
C:/Program Files/R/R-3.1.2/bin/x64/RGui.exe

Students will be required to install and to become proficient with the *RStudio* Integrated Development Environment (*IDE*),

http://www.rstudio.com/products/rstudio/

Using *RStudio*



Environments in R

Environments consist of a *frame* (a set of symbol-value pairs) and an *enclosure* (a pointer to an enclosing environment),

There are three system environments:

- `globalenv()` the user's workspace,
- `baseenv()` the environment of the base package,
- `emptyenv()` the only environment without an enclosure,

Environments form a tree structure of successive enclosures, with the empty environment at its root,

Packages have their own environments,

The enclosure of the base package is the empty environment,

```
> # get base environment
> baseenv()
> # get global environment
> globalenv()
> # get current environment
> environment()
> # get environment class
> class(environment())
> # define variable in current environment
> glob_var <- 1
> # get objects in current environment
> ls(environment())
> # create new environment
> new_env <- new.env()
> # get calling environment of new environment
> parent.env(new_env)
> # assign Value to Name
> assign("new_var1", 3, envir=new_env)
> # create object in new environment
> new_env$new_var2 <- 11
> # get objects in new environment
> ls(new_env)
> # get objects in current environment
> ls(environment())
> # environments are subset like lists
> new_env$new_var1
> # environments are subset like lists
> new_env[["new_var1"]]
```

The R Search Path

R evaluates variables using the search path, a series of environments:

- global environment,
- package environments,
- base environment,

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

The function `attach()` attaches objects to the search path,

Using `attach()` allows referencing object components by their names alone, rather than as components of objects,

The function `detach()` detaches objects from the search path,

The function `find()` finds where objects are located on the search path,

```
> search()  # get search path for R objects
 [1] ".GlobalEnv"        "package:knitr"
 [3] "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
> my_list <-
+   list(flowers=c("rose", "daisy", "tulip"),
+        trees=c("pine", "oak", "maple"))
> my_list$trees
[1] "pine"  "oak"   "maple"
> attach(my_list)
> trees
[1] "pine"  "oak"   "maple"
> search()  # get search path for R objects
 [1] ".GlobalEnv"        "my_list"
 [3] "package:knitr"     "package:stats"
 [5] "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"
 [9] "package:methods"   "Autoloads"
[11] "package:base"
> detach(my_list)
> head(trees)  # "trees" is in datasets base pac
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
4  10.5     72   16.4
```

Extracting Time Series from Environments

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

A list of *xts* can be flattened into a single *xts* 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 extractor (accessor) functions `Ad()`, `Vo()`, etc., extract columns from *OHLC* data,

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

```
> library(HighFreq)  # load package HighFreq
> # ETF symbols
> sym_bols <- c("VTI", "VEU", "IEF", "VNQ")
> # extract and merge all data, subset by sym_bols
> price_s <- do.call(merge,
+   as.list(rutils::env_etf)[sym_bols])
> # extract and merge adjusted prices, subset by sym_bols
> price_s <- do.call(merge,
+   lapply(as.list(rutils::env_etf)[sym_bols], Ad))
> # same, but works only for OHLC series
> price_s <- do.call(merge,
+   eapply(rutils::env_etf, Ad)[sym_bols])
> # drop ".Adjusted" from colnames
> colnames(price_s) <-
+   sapply(colnames(price_s),
+     function(col_name)
+ strsplit(col_name, split="[.]")[[1]])[1, ]
> tail(price_s[, 1:2], 3)
> # which objects in global environment are class xts?
> unlist(eapply(globalenv(), is.xts))
>
> # save xts to csv file
> write.zoo(price_s,
+     file='etf_series.csv', sep=",")
> # copy price_s into env_etf and save to .RData file
> assign("price_s", price_s, envir=env_etf)
> save(env_etf, file='etf_data.RData')
```

Referencing Object Components Using with()

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

with() allows referencing object components by their names alone,

It's often better to use with() instead of attach(),

```
> # "trees" is in datasets base package
> head(trees, 3)
  Girth Height Volume
1   8.3     70   10.3
2   8.6     65   10.3
3   8.8     63   10.2
> colnames(trees)
[1] "Girth"  "Height" "Volume"
> mean(Girth)

Error in mean(Girth):  object 'Girth' not
found

> mean(trees$Girth)
[1] 13.2
> with(trees,
+      c(mean(Girth), mean(Height), mean(Volume)))
[1] 13.2 76.0 30.2
```

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,

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.

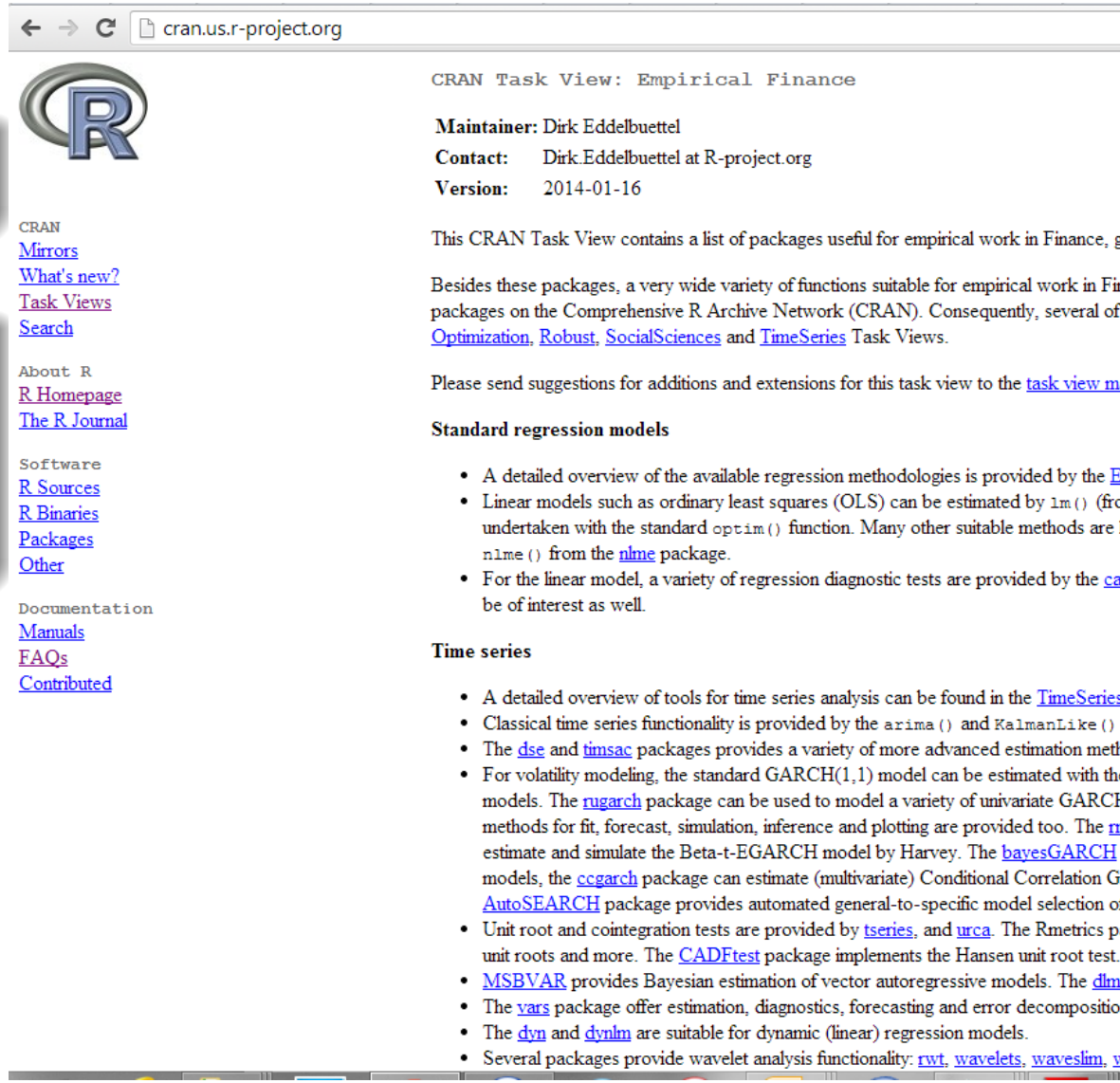
The package source code can be downloaded by clicking on the package source link,

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 sour
> .libPaths()  # get package save directory
```

```
> install.packages("AER")  # install "AER" from
> # install "PerformanceAnalytics" from R-Forge
> install.packages(
+   pkgs="PerformanceAnalytics",  # name
+   lib="C:/Users/Jerzy/Downloads",  # directory
+   repos="http://R-Forge.R-project.org")  # sou
```

```
> # install devtools from CRAN
> install.packages("devtools")
> # load devtools
> library(devtools)
> # install package "babynames" from GitHub
> install_github(repo="hadley/babynames")
```

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 so
> install.packages("PortfolioAnalytics",
+   type="source",
+   repos="http://r-forge.r-project.org")
> # download files for package "PortfolioAnalyti
> download.packages(pkgs = "PortfolioAnalytics",
+   destdir = ".",  # download to cwd
+   type = "source",
+   repos="http://r-forge.r-project.org")
> # install "PortfolioAnalytics" from local tar
> install.packages(
+   "C:/Users/Jerzy/Downloads/PortfolioAnalytics
+   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")
> pack_info <- installed.packages()  # matrix of
> # get a few package names and their versions
> pack_info[sample(x=1:100, 5), c("Package", "Ve
> t(pack_info["xts", ])  # get info for package
```

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
> gsub(
+   "C:/Users/Jerzy/Documents/R/win-library/3.1"
+   "~",
+   list.dirs(
+     file.path(
+       .libPaths()[1],
+       "PortfolioAnalytics")))
 [1] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [2] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [3] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [4] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [5] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [6] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [7] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [8] "C:/Users/Jerzy/Documents/R/win-library/3.4
 [9] "C:/Users/Jerzy/Documents/R/win-library/3.4
[10] "C:/Users/Jerzy/Documents/R/win-library/3.4
[11] "C:/Users/Jerzy/Documents/R/win-library/3.4
[12] "C:/Users/Jerzy/Documents/R/win-library/3.4
```

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 load
> library(MASS)
> # load package, return TRUE if loaded successf
> 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 succes
> 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
> v_wap <- TTR::VWAP(
+   price=quantmod::Ad(rutils::env_etf$VTI),
+   volume=quantmod::Vo(rutils::env_etf$VTI), n=
```

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 lists 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 th
> search()  # list all loaded packages on search
>
> # get documentation for package "Ecdat"
> packageDescription("Ecdat")  # get short descr
> help(package="Ecdat")  # load help page
> library(Ecdat)  # load package "Ecdat"
> data(package="Ecdat")  # list all datasets in
> ls("package:Ecdat")  # list all objects in "Ec
> browseVignettes("Ecdat")  # view package vigne
> detach("package:Ecdat")  # remove Ecdat from s
```

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

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
> detach("package:MASS")  # remove "MASS" from s
```

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

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")
```

Date Objects

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

The function as.Date() parses a character string into a date object,

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

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

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

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

```
> Sys.Date()  # get today's date
> date_time <- as.Date("2014-07-14")  # "%Y-%m-%d" or "%Y/
> date_time
> class(date_time)  # Date object
> as.Date("07-14-2014", "%m-%d-%Y")  # specify format
> date_time + 20  # add 20 days
> # extract internal representation to integer
> as.numeric(date_time)
> date_old <- as.Date("07/14/2013", "%m/%d/%Y")
> date_old
> # difference between dates
> difftime(date_time, date_old, units="weeks")
> weekdays(date_time)  # get day of the week
> # coerce numeric into date-times
> date_time <- 0
> attributes(date_time) <- list(class="Date")
> date_time  # "Date" object
> structure(0, class="Date")  # "Date" object
> structure(10000.25, class="Date")
```

POSIXct Date-time Objects

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

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

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

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

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

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

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

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

Operations on POSIXct Objects

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

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

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

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

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

```
> # same moment of time corresponds to different clock ti
> time_ny <- as.POSIXct("2014-07-14 13:30:10",
+       tz="America/New_York")
> time_ldn <- as.POSIXct("2014-07-14 13:30:10",
+       tz="UTC")
> # add five hours to POSIXct
> time_ny + 5*60*60
> # subtract POSIXct
> time_ny - time_ldn
> class(time_ny - time_ldn)
> # compare POSIXct
> time_ny > time_ldn
> # create vector of POSIXct times during trading hours
> trading_times <- seq(
+   from=as.POSIXct("2014-07-14 09:30:00", tz="America/Ne
+   to=as.POSIXct("2014-07-14 16:00:00", tz="America/New_
+   by="10 min")
> head(trading_times, 3)
> tail(trading_times, 3)
```

Moment of Time and Clock Time

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

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

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

```
> # POSIXct is stored as integer moment of time
> int_time <- as.numeric(date_time)
> # same moment of time corresponds to different clock ti
> as.POSIXct(int_time, origin="1970-01-01",
+       tz="America/New_York")
> as.POSIXct(int_time, origin="1970-01-01",
+       tz="UTC")
> # same clock time corresponds to different moments of t
> as.POSIXct("2014-07-14 13:30:10",
+       tz="America/New_York") -
+   as.POSIXct("2014-07-14 13:30:10", tz="UTC")
> # add 20 seconds to POSIXct
> date_time + 20
```

Methods for Manipulating POSIXct Objects

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

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

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

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

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

POSIXlt Date-time Objects

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

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

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

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

Adding a number to POSIXlt causes implicit coercion to POSIXct,

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

The methods round.POSIXt() and trunc.POSIXt() round and truncate

```
> # parse character string "%Y-%m-%d %H:%M:%S" to POSIXlt
> date_time <- as.POSIXlt("2014-07-14 18:30:10")
> date_time
> class(date_time)  # POSIXlt object
> as.POSIXct(date_time)  # coerce to POSIXct object
> # extract internal list representation to vector
> unlist(date_time)
> date_time + 20  # add 20 seconds
> class(date_time + 20)  # implicit coercion to POSIXct
> trunc(date_time, units="hours")  # truncate to closest h
> trunc(date_time, units="days")  # truncate to closest da
> methods(trunc)  # trunc methods
> trunc.POSIXt
```

Time Zones and Date-time Conversion

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

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

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

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

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

```
> Sys.timezone()  # get time-zone
> Sys.setenv(TZ="UTC")  # set time-zone to UTC
> Sys.timezone()  # get time-zone
> # Standard Time in effect
> as.POSIXct("2013-03-09 11:00:00", tz="America/New_York")
> # Daylight Savings Time in effect
> as.POSIXct("2013-03-10 11:00:00", tz="America/New_York")
> date_time <- Sys.time()  # today's date and time
> # convert to character in different TZ
> format(date_time, tz="America/New_York")
> format(date_time, tz="UTC")
> # parse back to POSIXct
> as.POSIXct(format(date_time, tz="America/New_York"))
> # difference between New_York time and UTC
> as.POSIXct(format(Sys.time(), tz="UTC")) -
+   as.POSIXct(format(Sys.time(), tz="America/New_York"))
> # set time-zone to New York
> Sys.setenv(TZ="America/New_York")
```

Manipulating Date-time Objects Using *lubridate*

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

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

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

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

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

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

Time Zones Using *lubridate*

lubridate simplifies *time zone* calculations,

lubridate uses the *UTC time zone* as default,

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

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

```
> date_time <- ymd_hms(20140714142010,
+                      tz="America/New_York")
> date_time
>
> # get same moment of time in "UTC" time zone
> with_tz(date_time, "UTC")
> as.POSIXct(format(date_time, tz="UTC"), tz="UTC")
>
> # get same clock time in "UTC" time zone
> force_tz(date_time, "UTC")
> as.POSIXct(format(date_time, tz="America/New_York"),
+      tz="UTC")
>
> # same moment of time
> date_time - with_tz(date_time, "UTC")
>
> # different moments of time
> date_time - force_tz(date_time, "UTC")
```

lubridate Time Span Objects

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

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

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

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

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

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

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

Adding Time Spans to Date-time Objects

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

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

End-of-month Dates

Adding monthly periods can create invalid dates,

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

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

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

Package *RQuantLib* Calendar Functions

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

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

```
> library(RQuantLib)  # load RQuantLib
>
> # create daily date series of class 'Date'
> in_dex <- Sys.Date() + -5:2
> in_dex
>
> # create Boolean vector of business days
> is_busday <- isBusinessDay(  # RQuantLib calendar
+   calendar="UnitedStates/GovernmentBond", in_dex)
>
> # create daily series of business days
> bus_index <- in_dex[is_busday]
> bus_index
```

Review of Date-time Classes in R

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

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

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

```
> date_time <- Sys.Date()  # create date series of class 'Date'
> in_dex <- date_time + 0:365  # daily series over one year
> head(in_dex, 4)  # print first few dates
> format(head(in_dex, 4), "%m/%d/%Y")  # print first few dates
> # create daily date-time series of class 'POSIXct'
> in_dex <- seq(Sys.time(), by="days", length.out=365)
> head(in_dex, 4)  # print first few dates
> format(head(in_dex, 4), "%m/%d/%Y %H:%M:%S")  # print first few d
> # create series of monthly dates of class 'zoo'
> monthly_index <- yearmon(2010+0:36/12)
> head(monthly_index, 4)  # print first few dates
> # create series of quarterly dates of class 'zoo'
> qrtly_index <- yearqtr(2010+0:16/4)
> head(qrtly_index, 4)  # print first few dates
> # parse quarterly 'zoo' dates to POSIXct
> Sys.setenv(TZ="UTC")
> as.POSIXct(head(qrtly_index, 4))
```

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

```
> # create daily time series ending today
> start_date <- decimal_date(Sys.Date()-6)
> end_date <- decimal_date(Sys.Date())
> # create vector of geometric Brownian motion
> da_ta <- exp(cumsum(rnorm(6)/100))
> fre_quency <- length(da_ta)/(end_date-start_da
> ts_series <- ts(data=da_ta,
+           start=start_date,
+           frequency=fre_quency)
> ts_series  # display time series
> # display index dates
> as.Date(date_decimal(coredata(time(ts_series))
> # bi-monthly geometric Brownian motion startin
> ts_series <- 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,

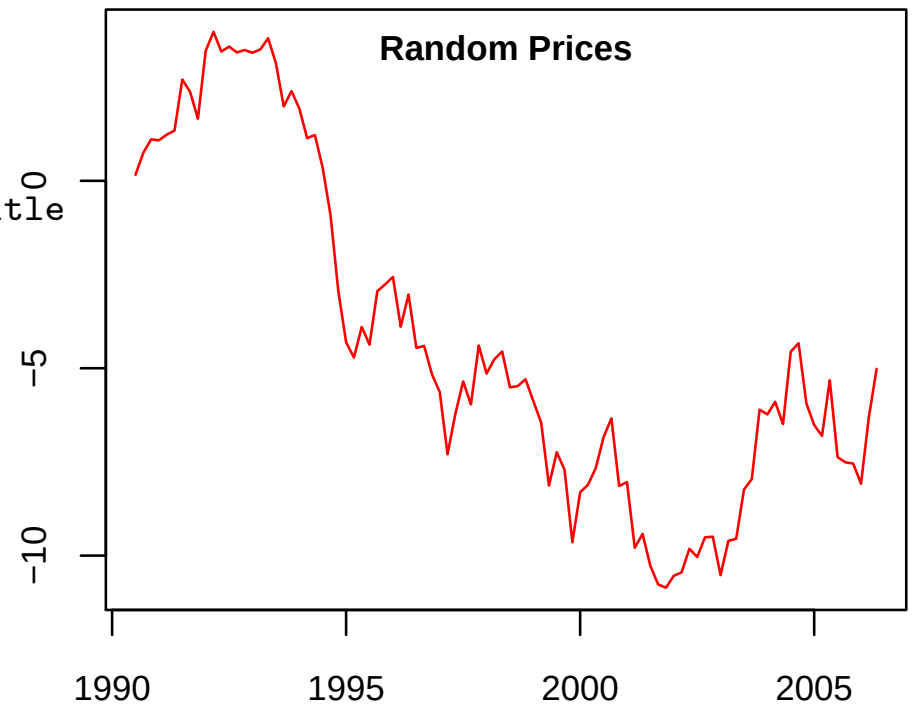
The function `window()` subsets the a *ts* time series object,

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

Plotting *ts* Time Series Objects

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

```
> plot(ts_series, type="l",  # create plot
+      col="red", lty="solid", xlab="", ylab="")
> title(main="Random Prices", line=-1)  # add title
```



EuStockMarkets Data

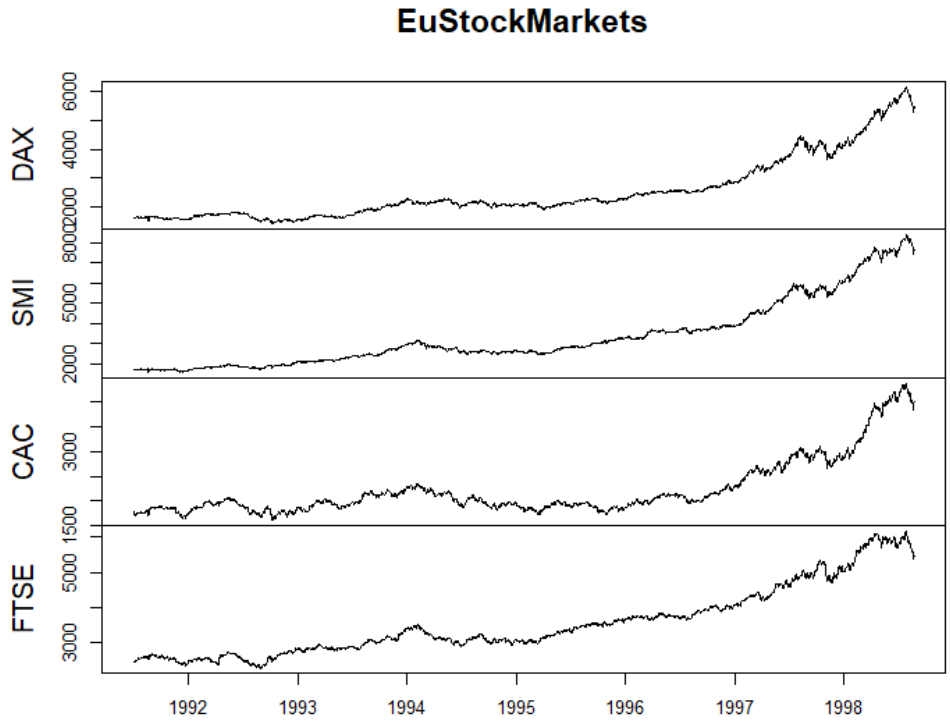
R includes a number of base packages that are already installed and loaded,

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 row
> # EuStockMarkets index is equally spaced
> diff(tail(time(EuStockMarkets), 11))
```

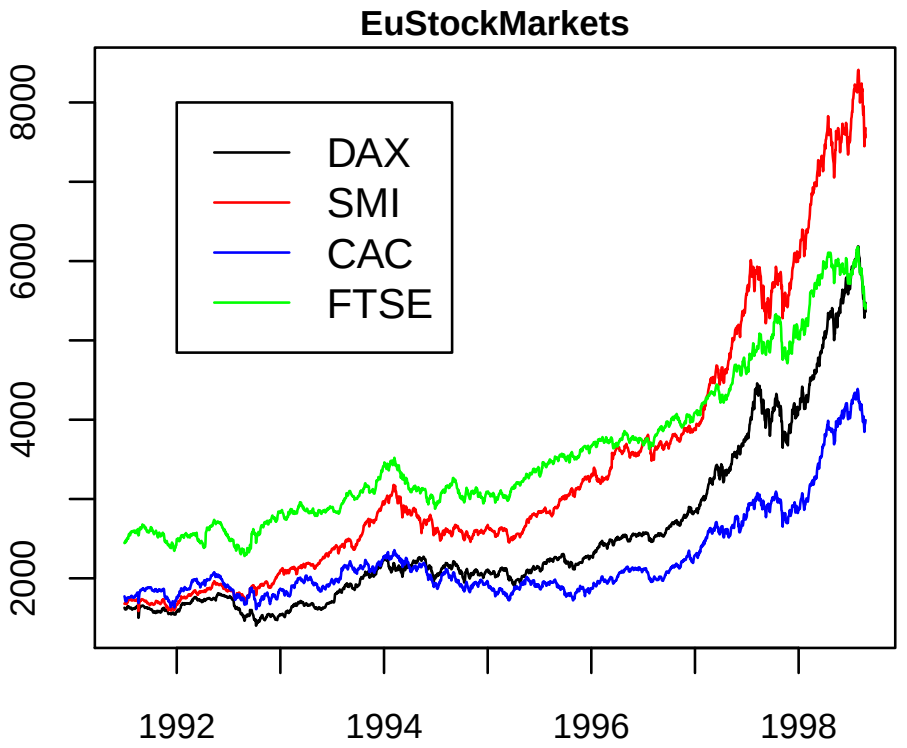
```
> # plot all the columns in separate panels
> plot(EuStockMarkets, main="EuStockMarkets", xlab="")
```

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)
```



Distribution of EuStockMarkets Data Returns

The function `hist()` calculates and plots a histogram, and returns its data invisibly,

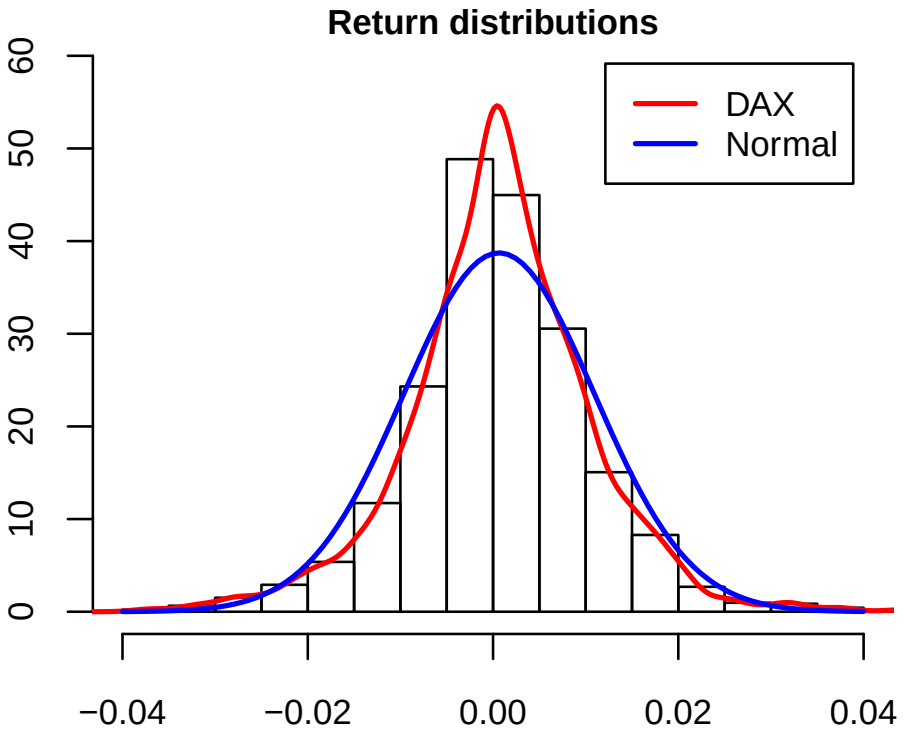
The parameter `breaks` is the number of cells of the histogram,

The function `density()` calculates a kernel estimate of the probability density,

The function `lines()` draws a line through specified points,

```
> # calculate DAX percentage returns
> dax_rets <- diff(log(EuStockMarkets[, 1]))
> # mean and standard deviation of returns
> c(mean(dax_rets), sd(dax_rets))
[1] 0.000652 0.010301
```

```
> # plot histogram
> hist(dax_rets, breaks=30, main="",
+      xlim=c(-0.04, 0.04), ylim=c(0, 60),
+      xlab="", ylab="", freq=FALSE)
> # draw kernel density of histogram
> lines(density(dax_rets), col='red', lwd=2)
> # add density of normal distribution
> curve(expr=dnorm(x, mean=mean(dax_rets), sd=sd(dax_rets)),
+ add=TRUE, type="l", lwd=2, col="blue")
> title(main="Return distributions", line=0)  # add title
> # add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
```



EuStockMarkets Data Quantile-Quantile Plot

A Q-Q plot is a plot of points with the same quantiles, from two probability distributions,

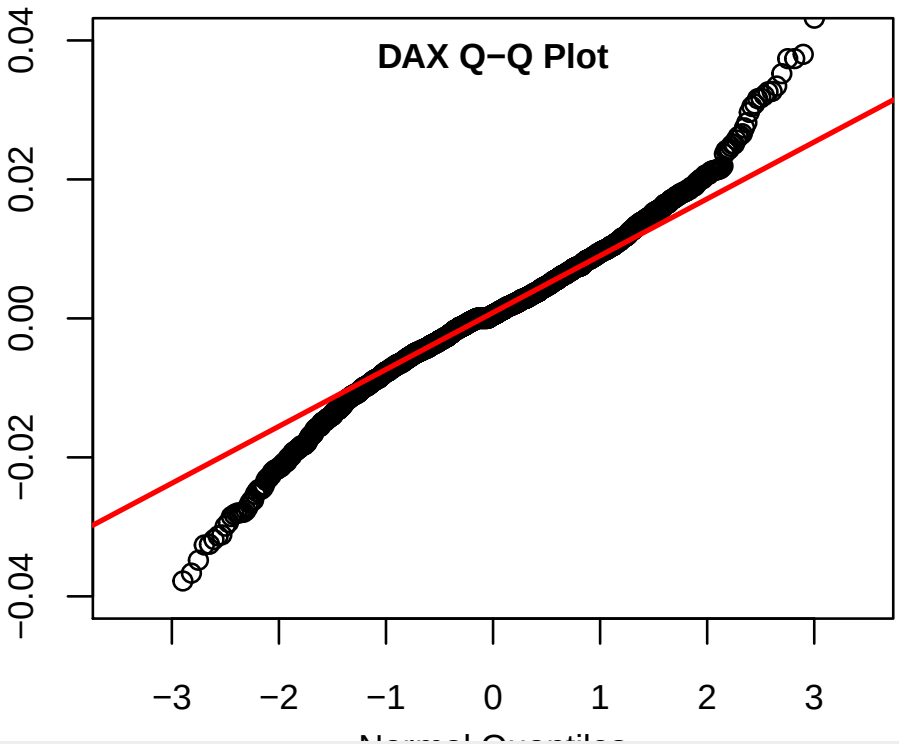
The function qqnorm() produces a normal Q-Q plot,

The function qqline() fits a line to the normal quantiles,

The *p*-value of the *Shapiro-Wilk* test is very close to zero, which shows that the DAX returns are very unlikely to be normal,

```
> # calculate percentage returns
> dax_rets <- diff(log(EuStockMarkets[, 1]))
> # perform Shapiro-Wilk test
> shapiro.test(dax_rets)
```

The DAX Q-Q plot shows that the DAX return distribution has fat tails,




```
> # create normal Q-Q plot
> qqnorm(dax_rets, ylim=c(-0.04, 0.04),
+  xlab='Normal Quantiles', main='')
> # fit a line to the normal quantiles
> qqline(dax_rets, col='red', lwd=2)
> plot_title <- paste(colnames(EuStockMarkets)[1
+              'Q-Q Plot')
> title(main=plot_title, line=-1)  # add title
```

Boxplots of Distributions of Values

Box-and-whisker plots (*boxplots*) are graphical representations of a distribution of values,

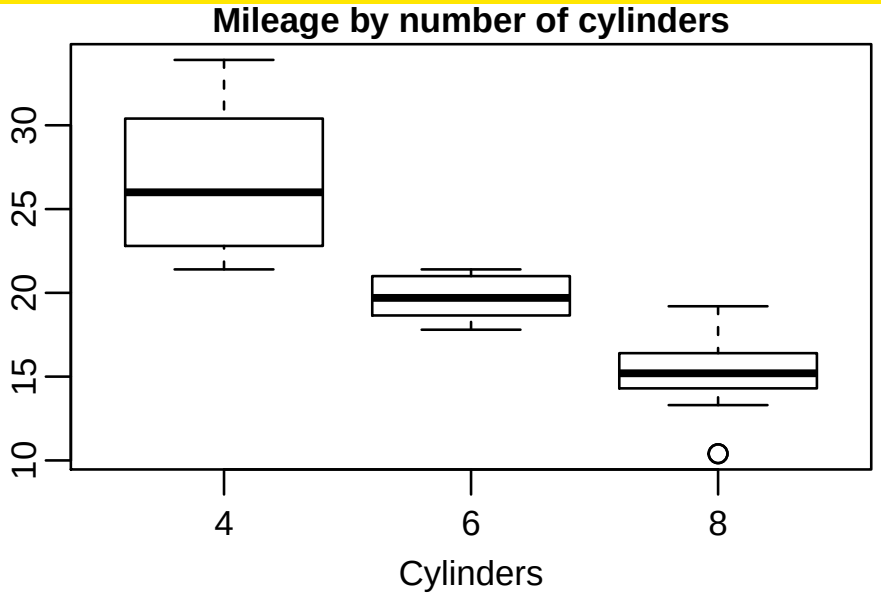
The bottom and top box edges (*hinges*) are equal to the first and third quartiles, and the *box* width is equal to the interquartile range (*IQR*),

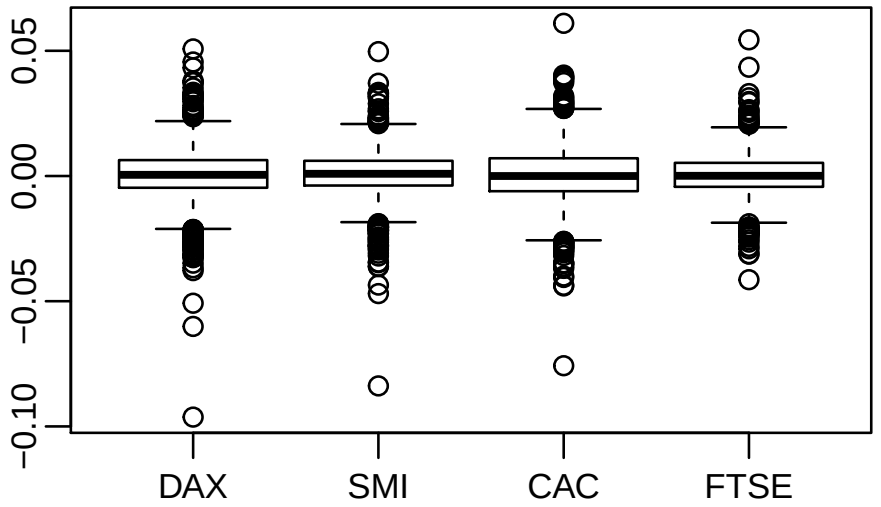
The nominal range is equal to 1.5 times the *IQR* above and below the box *hinges*,

The *whiskers* are dashed vertical lines representing values beyond the first and third quartiles, but within the nominal range,

The *whiskers* end at the last values within the nominal range, while the open circles represent outlier values beyond the nominal range,

The function `boxplot()` has two `methods`: one for `formula` objects (for categorical variables), and another for `data frames`,






```
> # boxplot method for formula
> boxplot(formula=mpg ~ cyl, data=mtcars,
+   main="Mileage by number of cylinders",
+   xlab="Cylinders", ylab="Miles per gallon")
> # boxplot method for data frame of EuStockMarkets percentage returns
> boxplot(x=diff(log(EuStockMarkets)))
```

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* indeices 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,

```
> # create zoo time series of random returns
> in_dex <- Sys.Date() + 0:3
> zoo_series <- zoo(rnorm(length(in_dex)),
+           order.by=in_dex)
> zoo_series
> attributes(zoo_series)
> class(zoo_series)  # class 'zoo'
> tail(zoo_series, 3)  # get last few elements
```

Operations on *zoo* Time Series

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

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

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,

```
> coredata(zoo_series)  # extract coredata
> index(zoo_series)  # extract time index
> start(zoo_series)  # first date
> end(zoo_series)  # last date
> zoo_series[start(zoo_series)]  # first element
> zoo_series[end(zoo_series)]  # last element
> coredata(zoo_series) <- rep(1, 4)  # replace c
> cumsum(zoo_series)  # cumulative sum
> cummax(cumsum(zoo_series))
> cummin(cumsum(zoo_series))
```

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,

```
> zoo_series <-
+   zoo(as.matrix(cumsum(rnorm(100)), nc=1),
+ order.by=seq(from=as.Date("2013-06-15"),
+              by="day", length.out=100))
> colnames(zoo_series) <- "zoo_series"
> tail(zoo_series)
> dim(zoo_series)
> attributes(zoo_series)
```

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 present 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,

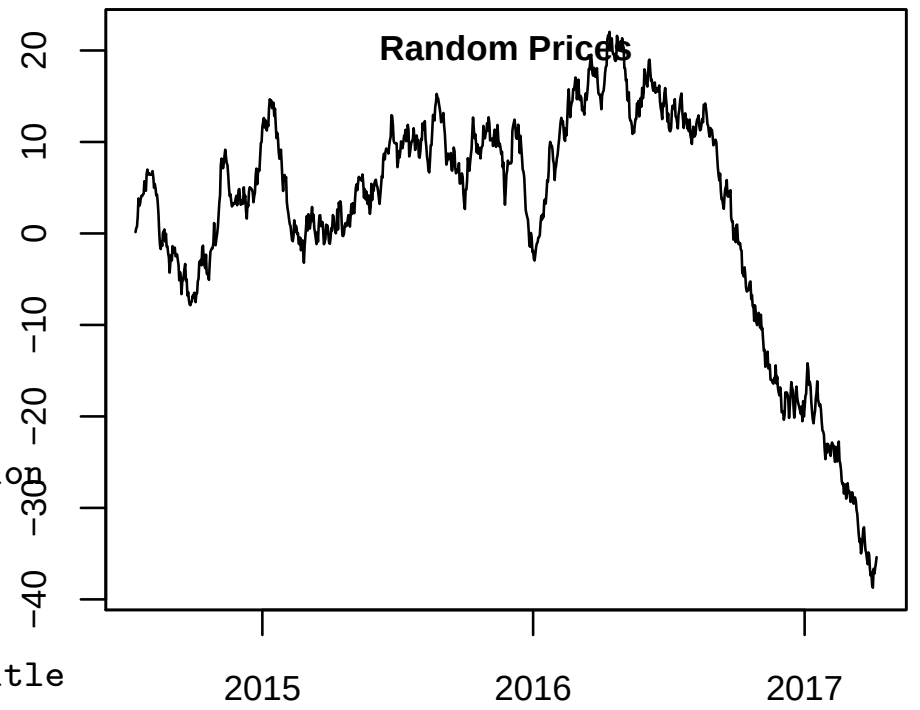
```
> coredata(zoo_series) <- (1:4)^2  # replace cor
> zoo_series
> lag(zoo_series)  # one day lag
> lag(zoo_series, 2)  # two day lag
> lag(zoo_series, k=-1)  # proper one day lag
> diff(zoo_series)  # diff with one day lag
> # proper lag and original length
> lag(zoo_series, -2, na.pad=TRUE)
```

Plotting *zoo* Time Series

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

```
> library(zoo)  # load package zoo
> # create index of daily dates
> in_dex <- seq(from=as.Date("2014-07-14"),
+               by="day", length.out=1000)
> # create vector of geometric Brownian motion
> zoo_data <-
+   exp(cumsum(rnorm(length(in_dex))/100))
> # create zoo series of geometric Brownian motion
> zoo_series <- zoo(x=zoo_data, order.by=in_dex)
```

```
> # plot using plot.zoo method
> plot(zoo_series, xlab="", ylab="")
> title(main="Random Prices", 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,

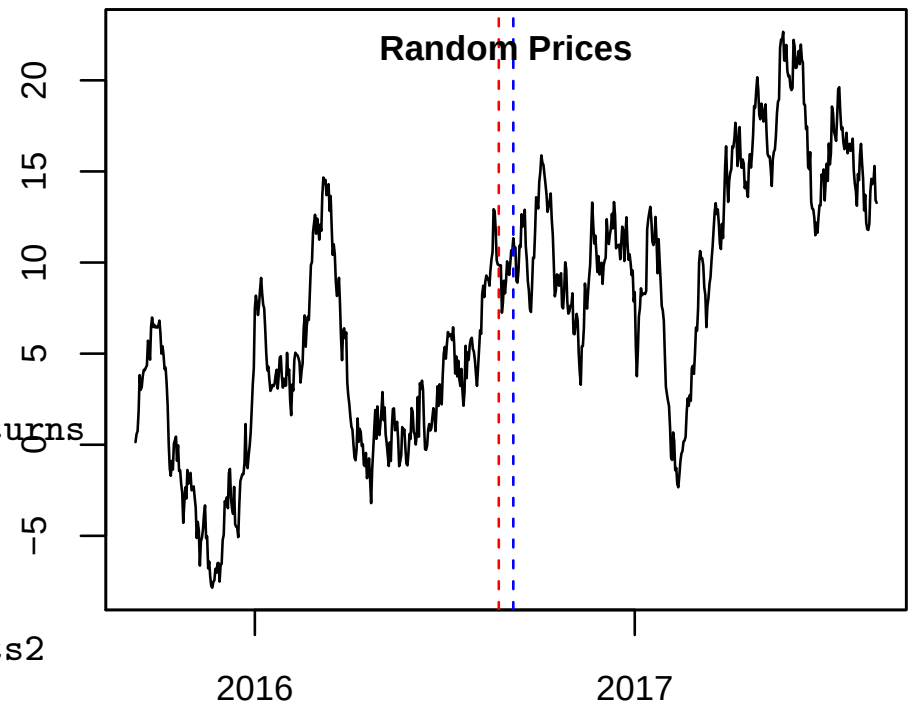
In addition, *zoo* time series can be subset using `Date` objects,

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

Sequential Joining *zoo* Time Series

zoo time series can be joined sequentially using function `rbind()`,

```
> library(zoo)  # load package zoo
> # create daily date series of class 'Date'
> in_dex1 <- seq(Sys.Date(), by="days",
+                length.out=365)
> # create zoo time series of random returns
> zoo_series1 <- zoo(rnorm(length(in_dex1)),
+             order.by=in_dex1)
> # create another zoo time series of random returns
> in_dex2 <- seq(Sys.Date()+350, by="days",
+                length.out=365)
> zoo_series2 <- zoo(rnorm(length(in_dex2)),
+             order.by=in_dex2)
> # rbind the two time series - ts1 supersedes ts2
> zoo_series3 <- rbind(zoo_series1,
+             zoo_series2[index(zoo_series2) > end(zoo_series1)])
> # plot zoo time series of geometric Brownian motion
> plot(exp(cumsum(zoo_series3)/100), xlab="", ylab="")
> # add vertical lines at stitch point
> abline(v=end(zoo_series1), col="blue", lty="dashed")
> abline(v=start(zoo_series2), col="red", lty="dashed")
> title(main="Random Prices", 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'
> in_dex1 <- Sys.Date() + -3:1
> # create zoo time series of random returns
> zoo_series1 <- zoo(rnorm(length(in_dex1)),
+           order.by=in_dex1)
> # create another zoo time series of random ret
> in_dex2 <- Sys.Date() + -1:3
> zoo_series2 <- zoo(rnorm(length(in_dex2)),
+           order.by=in_dex2)
> merge(zoo_series1, zoo_series2)  # union of da
> # intersection of dates
> merge(zoo_series1, zoo_series2, 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() copies the last non-NA values onto future NA values (*locf* stands for *last observation carry forward*),

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

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

Initial NA prices should be initialized to the first non-NA price, which can be done by calling zoo::na.locf() with the argument fromLast=TRUE,

Initial NA returns should be initialized to zero,

```
> # create matrix containing NA values
> mat_rix <- sample(18)
> mat_rix[sample(NROW(mat_rix), 4)] <- NA
> mat_rix <- matrix(mat_rix, nc=3)
> # replace NA values with most recent non-NA va
> zoo::na.locf(mat_rix)
> rutils::na_locf(mat_rix)
> # get time series of prices
> price_s <- mget(c("VTI", "VXX"), envir=rutils:
> price_s <- lapply(price_s, quantmod::Ad)
> price_s <- rutils::do_call(cbind, price_s)
> sum(is.na(price_s))
> # carry forward and backward non-NA prices
> price_s <- zoo::na.locf(price_s)
> price_s <- zoo::na.locf(price_s, fromLast=TRUE
> sum(is.na(price_s))
> # remove whole rows containing NA returns
> re_turns <- rutils::env_etf$re_turns
> sum(is.na(re_turns))
> re_turns <- na.omit(re_turns)
> # or carry forward non-NA returns (preferred)
> re_turns <- rutils::env_etf$re_turns
> re_turns[1, is.na(re_turns[1, ])] <- 0
> re_turns <- zoo::na.locf(re_turns)
> sum(is.na(re_turns))
```

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
> zoo_series <- as.zoo(EuStockMarkets)
> class(index(zoo_series))  # index is numeric
> head(zoo_series, 3)
> # approximately convert index into class 'Date'
> index(zoo_series) <-
+   as.Date(365*(index(zoo_series)-1970))
> head(zoo_series, 3)
> # convert index into class 'POSIXct'
> zoo_series <- as.zoo(EuStockMarkets)
> index(zoo_series) <- date_decimal(index(zoo_series))
> head(zoo_series, 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,

```
> # create index of daily dates
> in_dex <- seq(from=as.Date("2014-07-14"),
+               by="day", length.out=1000)
> # create vector of geometric Brownian motion
> zoo_data <- exp(cumsum(rnorm(length(in_dex))/100))
> # create zoo time series of geometric Brownian motion
> zoo_series <- zoo(x=zoo_data,
+               order.by=in_dex)
> head(zoo_series, 3)  # zoo object
> # as.ts() creates ts object with frequency=1
> ts_series <- as.ts(zoo_series)
> tsp(ts_series)  # frequency=1
> # get start and end dates of zoo_series
> start_date <- decimal_date(start(zoo_series))
> end_date <- decimal_date(end(zoo_series))
> # calculate frequency of zoo_series
> fre_quency <- length(zoo_series)/(end_date-start_date)
> da_ta <- coredata(zoo_series)  # extract data from zoo_s
> # create ts object using ts()
> ts_series <- ts(data=da_ta, start=start_date,
+           frequency=fre_quency)
> # display start of time series
> window(ts_series, start=start(ts_series),
+  end=start(ts_series)+4/365)
> head(time(ts_series))  # display index dates
> head(as.Date(date_decimal(coredata(time(ts_series)))))
```

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
> week_days <- weekdays(index(zoo_series))
> is_weekday <- !((week_days == "Saturday") |
+   (week_days == "Sunday"))
> # remove weekends from zoo time series
> zoo_series <- zoo_series[is_weekday, ]
> head(zoo_series, 7)  # zoo object
> # as.ts() creates NA values
> ts_series <- as.ts(zoo_series)
> head(ts_series, 7)
> # create vector of regular dates, including weekends
> in_dex <- seq(from=start(zoo_series),
+               by="day",
+               length.out=length(zoo_series))
> index(zoo_series) <- in_dex
> ts_series <- as.ts(zoo_series)
> head(ts_series, 7)
```

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,

```
> library(xts)  # load package xts
> # create xts time series of random returns
> in_dex <- Sys.Date() + 0:3
> x_ts <- xts(rnorm(length(in_dex)),
+             order.by=in_dex)
> names(x_ts) <- "random"
> x_ts
> tail(x_ts, 3)  # get last few elements
> first(x_ts)  # get first element
> last(x_ts)  # get last element
> class(x_ts)  # class 'xts'
> attributes(x_ts)
> # get the time zone of an xts object
> indexTZ(x_ts)
```

Coercing *zoo* Time Series Into Class *xts*

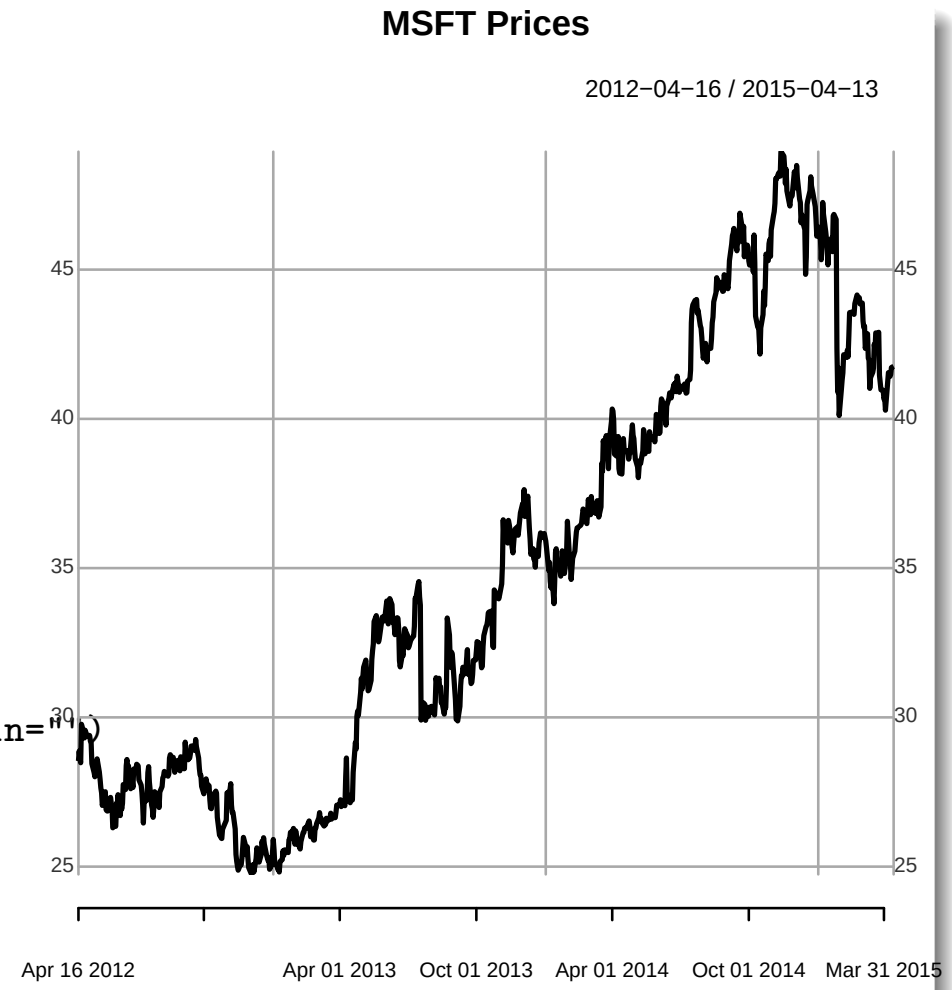
The function `as.xts()` coerces time series (including *zoo*) into *xts* time series,

`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,

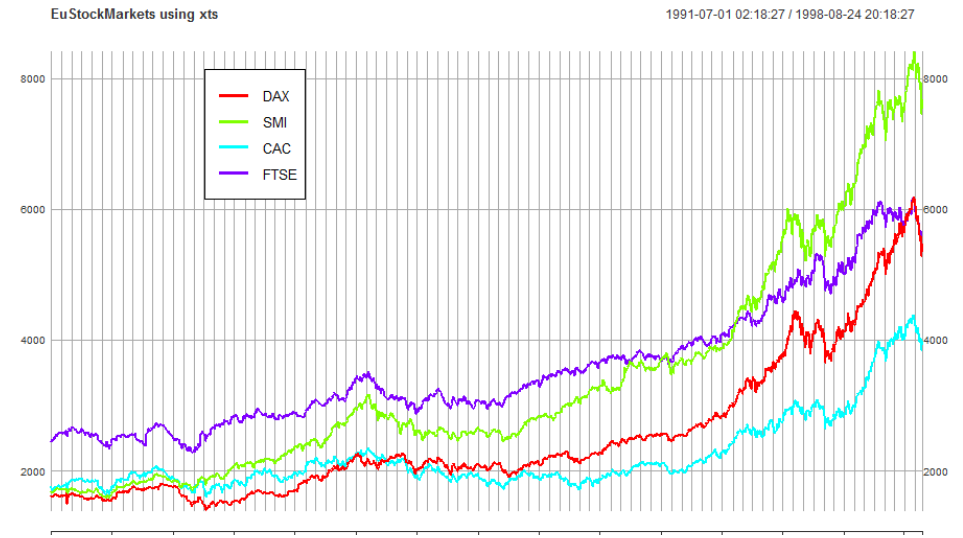
```
> library(xts)  # load package xts
> # as.xts() creates xts from zoo
> st_ox <- as.xts(zoo_stx_adj)
> dim(st_ox)
> head(st_ox[, 1:4], 4)
```

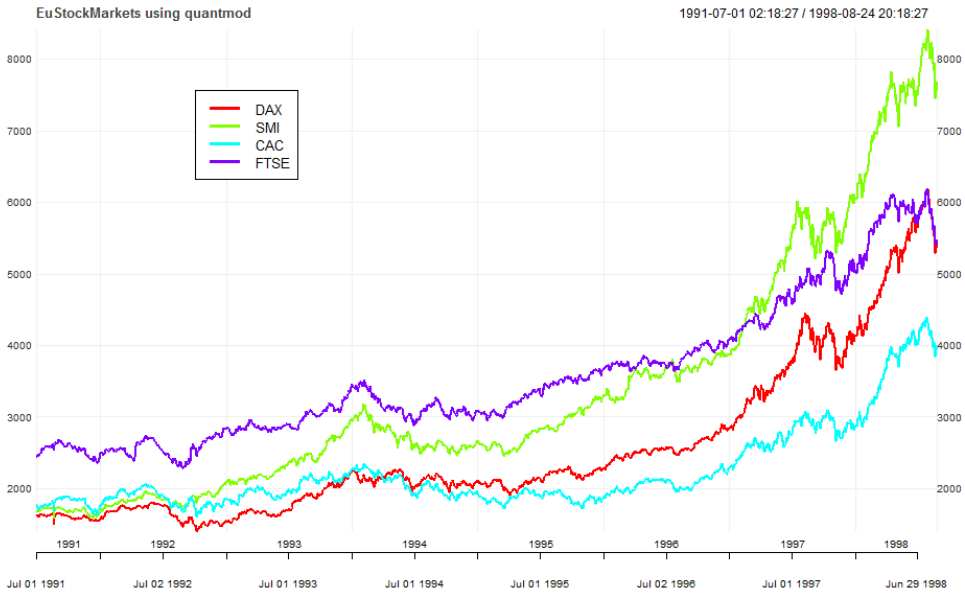
```
> # plot using plot.xts method
> plot(st_ox[, "AdjClose"], xlab="", ylab="", main="")
> title(main="MSFT Prices")  # add title
```



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

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





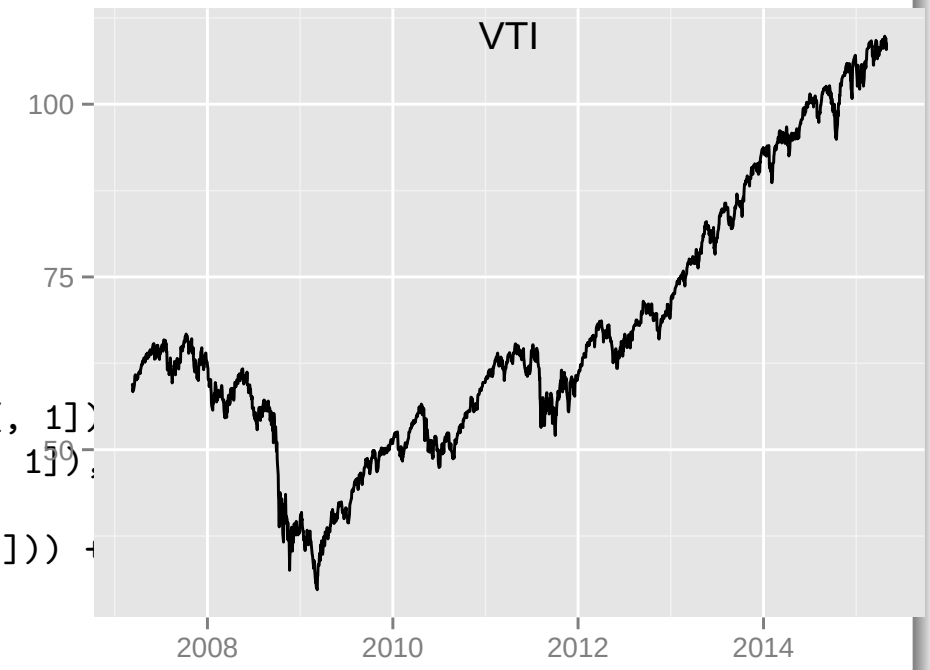
Plotting *xts* Using Package *ggplot2*

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)
> # create ggplot object
> etf_gg <- qplot(x=index(rutils::env_etf$price_s[, 1]),
+           y=as.numeric(rutils::env_etf$price_s[, 1]),
+           geom="line",
+           main=names(rutils::env_etf$price_s[, 1])) +
+   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
> etf_gg
```

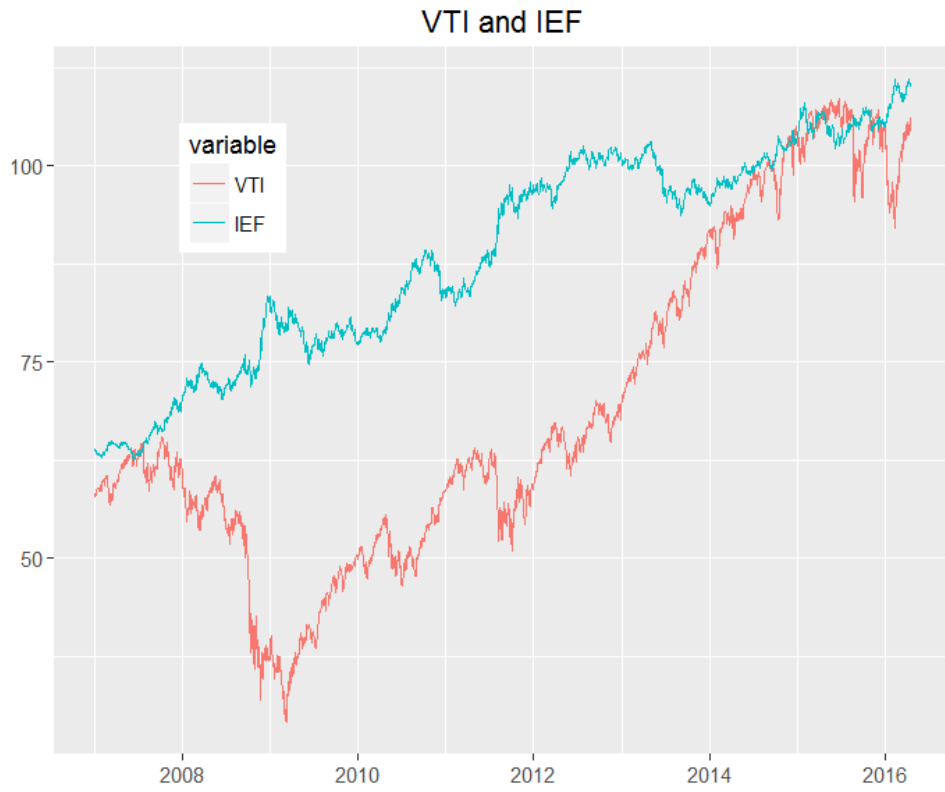


Plotting Multiple *xts* Using Package *ggplot2*

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 formatted into `data frames`,

```
> library(rutils)  # load xts time series data
> library(reshape2)
> library(ggplot2)
> # create data frame of time series
> data_frame <-
+   data.frame(dates=index(rutils::env_etf$price
+     coredata(rutils::env_etf$price_s[, c("VTI"
> # reshape data into a single column
> data_frame <-
+   reshape2::melt(data_frame, id="dates")
> x11(width=6, height=5)  # open plot window
> # ggplot the melted data_frame
> ggplot(data=data_frame,
+  mapping=aes(x=dates, 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
```



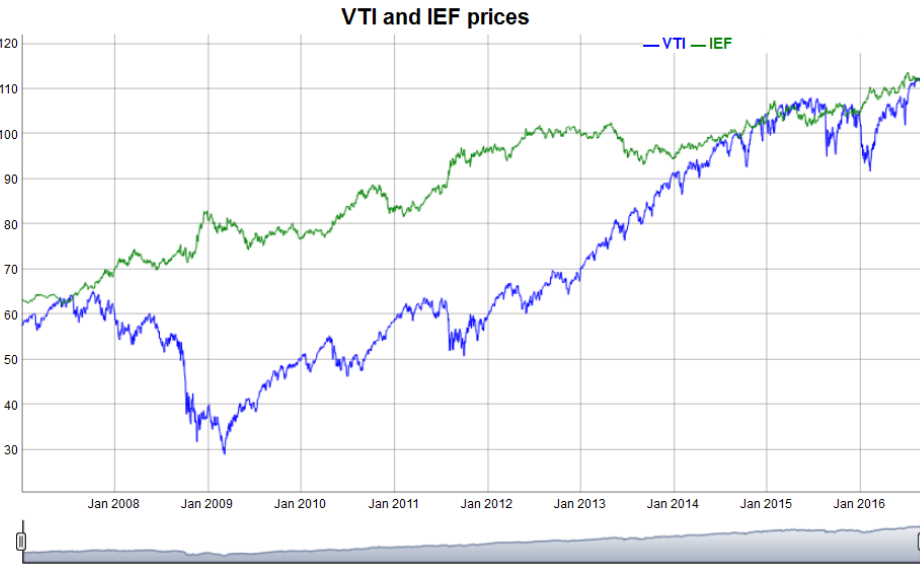
Time series with multiple columns must be reshaped into a single column, which can be performed using the function `melt()` from package *reshape2*,

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 env_etf dataset
> suppressMessages(suppressWarnings(library(ruti
> suppressMessages(suppressWarnings(library(dygraphs)))
> x_ts <- rutils::env_etf$price_s[, c("VTI", "IEF")]
> # plot dygraph with date range selector
> dygraph(x_ts, 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,

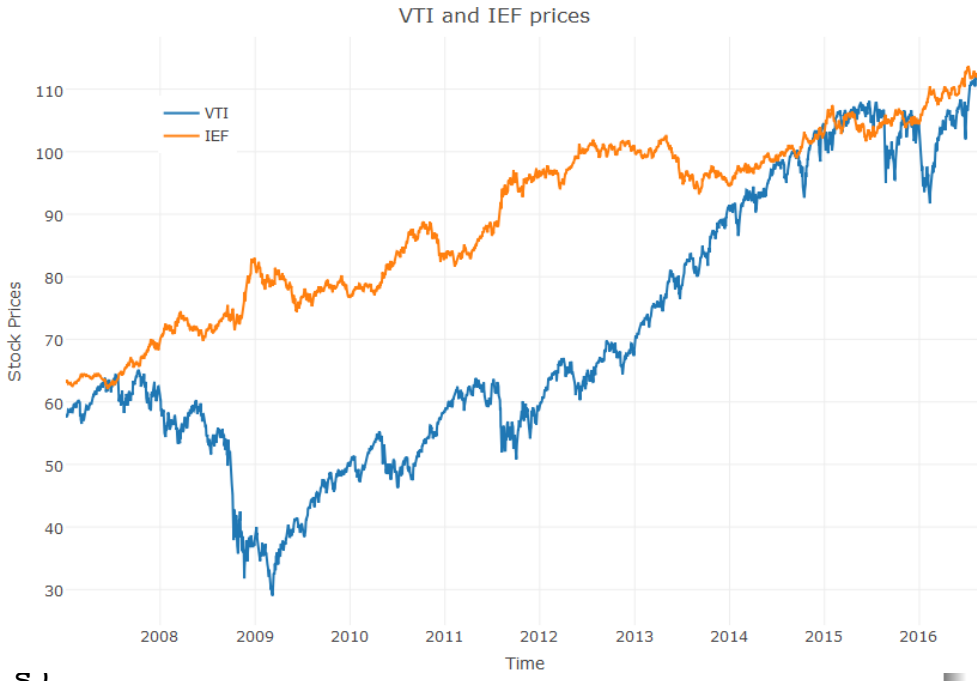
But *pdf* documents can't run *JavaScript* code, so they can't display interactive *dygraphs* plots,

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 env_etf dataset
> suppressMessages(suppressWarnings(library(ruti
> suppressMessages(suppressWarnings(library(plot
> # create data frame of time series
> data_frame <-
+   data.frame(dates=index(rutils::env_etf$price_s),
+     coredata(rutils::env_etf$price_s[, c("VTI", "IEF")]))
> # plotly syntax using pipes
> data_frame %>%
+   plot_ly(x=~dates, y=~VTI, type="scatter", mode="lines", name="VTI") %>%
+   add_trace(x=~dates, 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
> p_lot <- plot_ly(data=data_frame, x=~dates, y=~VTI, type="scatter", mode="lines", name="VTI")
> p_lot <- add_trace(p=p_lot, x=~dates, y=~IEF, type="scatter", mode="lines", name="IEF")
```

Subsetting *xts* Time Series

xts time series can be subset in similar ways to *zoo*,

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 much faster than the bracket `"[]"` notation,

```
> # subset xts using a date range string
> stox_sub <- st_ox["2014-10-15/2015-01-10", 1:4]
> first(stox_sub)
> last(stox_sub)
> # subset Nov 2014 using a date string
> stox_sub <- st_ox["2014-11", 1:4]
> first(stox_sub)
> last(stox_sub)
> # subset all data after Nov 2014
> stox_sub <- st_ox["2014-11/", 1:4]
> first(stox_sub)
> last(stox_sub)
> # comma after date range not necessary
> identical(st_ox["2014-11", ], st_ox["2014-11"])
> # benchmark the speed of subsetting
> library(microbenchmark)
> summary(microbenchmark(
+   bracket=sapply(500,
+   function(in_dex) max(st_ox[in_dex:(in_dex+10), ])),
+   subset=sapply(500,
+   function(in_dex) max(xts::.subset_xts(st_ox, in_dex:(
+   times=10))[, c(1, 4, 5)]
```

Subsetting Recurring *xts* Time Intervals

A *recurring time interval* is the same time interval every day,

xts 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"

```
> # vector of 1-minute times (ticks)
> min_ticks <- seq.POSIXt(
+   from=as.POSIXct("2015-04-14", tz="America/New_York"),
+   to=as.POSIXct("2015-04-16"),
+   by="min")
> # xts of 1-minute times (ticks) of random returns
> x_ts <- xts(rnorm(length(min_ticks)),
+               order.by=min_ticks)
> # subset recurring time interval using "T notation",
> x_ts <- x_ts["T09:30:00/T16:00:00"]
> first(x_ts["2015-04-15"])  # first element of day
> last(x_ts["2015-04-15"])  # last element of day
> # suppress timezone warning messages
> options(xts_check_tz=FALSE)
```

Properties of *xts* Time Series

xts series always have a `dim` attribute, unlike *zoo*,

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,

```
> str(st_ox)  # display structure of xts
> # subsetting zoo to single column drops dim attribute
> dim(zoo_stx_adj)
> dim(zoo_stx_adj[, 1])
> # zoo with single column are vectors not matrices
> c(is.matrix(zoo_stx_adj), is.matrix(zoo_stx_adj[, 1]))
> # xts always have a dim attribute
> rbind(base=dim(st_ox), subs=dim(st_ox[, 1]))
> c(is.matrix(st_ox), is.matrix(st_ox[, 1]))
```

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

`lag()` and `diff()` operations on *xts* series differ from those on *zoo*,

`lag()` and `diff()` operations on *zoo* series shorten the series by one row,

By default, the `lag()` operation on *xts* replaces the present value with values from the past (negative lags replace with values from the future),

By default, the `lag()` and `diff()` operations on *xts* retain the same number of rows, but substitute `NAs` for missing data,

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

Determining Calendar *Endpoints* 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 *endpoints* 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
> end_points <- endpoints(price_s, on="hours")
> head(end_points)
> # extract the last observations in each hour
> head(price_s[end_points, ])
```

Converting *xts* 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,

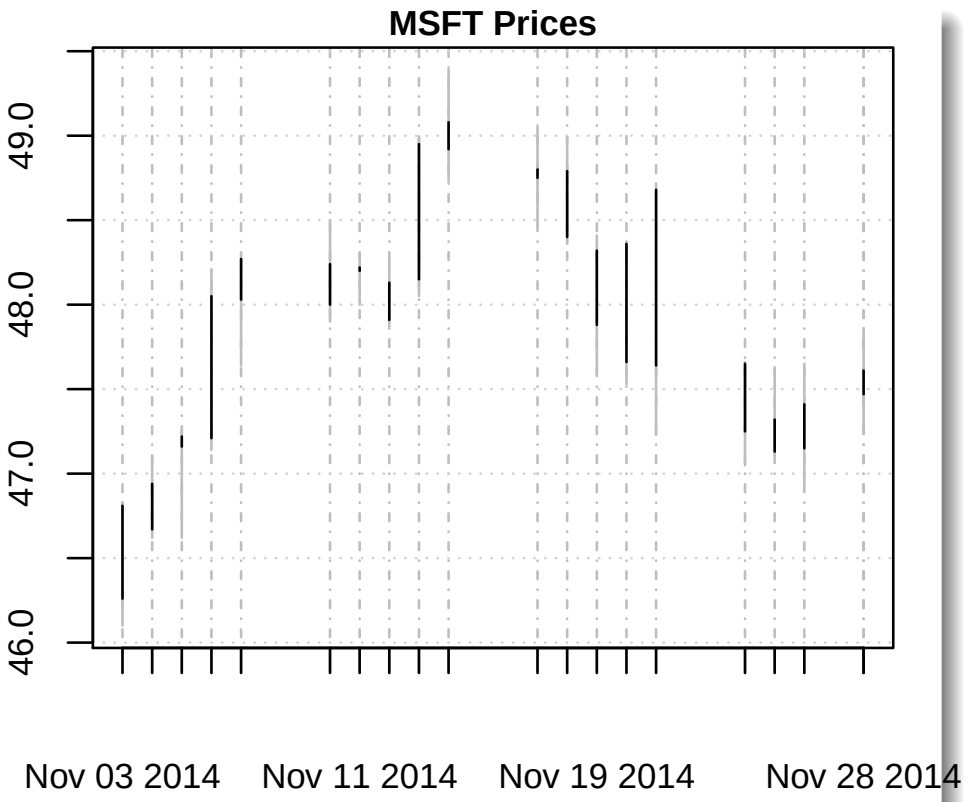
`to.period()` converts both univariate and *OHLC* time series to a lower periodicity,

```
> # lower the periodicity to months
> xts_monthly <- to.period(x=st_ox,
+                period="months", name="MSFT")
> # convert colnames to standard OHLC format
> colnames(xts_monthly)
> colnames(xts_monthly) <- sapply(
+   strsplit(colnames(xts_monthly), split=".", fixed=TRUE)
+   function(na_me) na_me[-1]
+   )  # end sapply
> head(xts_monthly, 3)
> # lower the periodicity to years
> xts_yearly <- to.period(x=xts_monthly,
+                period="years", name="MSFT")
> colnames(xts_yearly) <- sapply(
+   strsplit(colnames(xts_yearly), split=".", fixed=TRUE)
+   function(na_me) na_me[-1]
+   )  # end sapply
> head(xts_yearly)
```

Plotting *OHLC* Time Series Using *xts*

The method (function) `plot.xts()` can plot *OHLC* time series of class *xts*,

```
> library(xts)  # load package xts
> # as.xts() creates xts from zoo
> st_ox <- as.xts(zoo_stx_adj)
> # subset xts using a date
> stox_sub <- st_ox["2014-11", 1:4]
>
> # plot OHLC using plot.xts method
> plot(stox_sub, type="candles", main="")
> title(main="MSFT Prices")  # add title
```



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,

```
> ts_stx <- as.ts(zoo_stx)
> class(ts_stx)
> tail(ts_stx[, 1:4])
> library(xts)
> st_ox <- as.xts(zoo_stx)
> class(st_ox)
> tail(st_ox[, 1:4])
```

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*,
- fitting models to time series,
- portfolio optimization,

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

```
> # get documentation for package tseries
> packageDescription("tseries")  # get short des
>
> help(package="tseries")  # load help page
>
> library(tseries)  # load package tseries
>
> data(package="tseries")  # list all datasets i
>
> ls("package:tseries")  # list all objects in "
>
> detach("package:tseries")  # remove tseries fr
```

Downloading ts Data From *Yahoo* 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,

```
> # download MSFT data in ts format
> ts_stx <- 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
> adj_vector <-
+   as.vector(ts_stx[, "AdjClose"] / ts_stx[, "Close"])
> # adjust OHLC prices
> ts_stx_adj <- ts_stx
> ts_stx_adj[, c("Open","High","Low","Close")] <-
+   adj_vector * ts_stx[, c("Open","High","Low","Close")]
> # inspect the data
> tsp(ts_stx_adj)  # frequency=1
> head(time(ts_stx_adj))
> head(ts_stx_adj)
> tail(ts_stx_adj)
```

Downloading *zoo* Data From *Yahoo* 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
> zoo_stx <- 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(zoo_stx)
> dim(zoo_stx)
> head(zoo_stx, 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
> adj_vector <-
+   as.vector(zoo_stx[, "AdjClose"] / zoo_stx[, "Close"])
> head(adj_vector, 5)
> tail(adj_vector, 5)
> # adjust OHLC prices
> zoo_stx_adj <- zoo_stx
> zoo_stx_adj[, c("Open","High","Low","Close")] <-
+   adj_vector * zoo_stx[, c("Open","High","Low","Close")]
> head(zoo_stx_adj)
> tail(zoo_stx_adj)
```

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")`,

```
> # download EUR/USD data
> zoo_eurusd <- 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
> zoo_stxeur <- cbind(zoo_eurusd,
+                     zoo_stx[, "AdjClose"])
> colnames(zoo_stxeur) <- c("EURUSD", "MSFT")
> zoo_stxeur <-
+   zoo_stxeur[complete.cases(zoo_stxeur),]
> save(zoo_stx, zoo_stx_adj,
+      ts_stx, ts_stx_adj,
+      zoo_eurusd, zoo_stxeur,
+      file="C:/Develop/R/lecture_slides/data/zoo_data.RDa
```

```
> # inspect the data
> class(zoo_eurusd)
> tail(zoo_eurusd, 4)
```

Downloading Data for Multiple Symbols Using *tseries*

Data for multiple symbols can be downloaded in an `lapply` loop,

`lapply` returns a *list* of *zoo* objects,

The list of *zoo* objects can be flattened into a single *zoo* object 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,

```
> # ETF symbols for asset allocation
> sym_bols <- c("VTI", "VEU", "IEF", "VNQ",
+   "DBC", "VXX", "XLY", "XLP", "XLE", "XLF",
+   "XLV", "XLI", "XLB", "XLK", "XLU", "VYM",
+   "IVW", "IWB", "IWD", "IWF")
> # download price and volume data for sym_bols into list
> zoo_series <- suppressWarnings(
+   lapply(sym_bols, # loop for loading data
+    get.hist.quote,
+    quote=c("AdjClose", "Volume"),
+    start=Sys.Date()-3650,
+    end=Sys.Date(),
+    origin="1970-01-01")
+ )  # end suppressWarnings
> # flatten list of zoo objects into a single zoo object
> zoo_series <- do.call(merge, zoo_series)
> # or
> zoo_series <- rutils::do_call(cbind, zoo_series)
> # assign names in format "symbol.Close", "symbol.Volume"
> names(zoo_series) <-
+   as.vector(sapply(sym_bols,
+                    paste, c("Close", "Volume"), sep="."))
> # save zoo_series to a comma-separated CSV file
> write.zoo(zoo_series, file='zoo_series.csv', sep=",")
> # save zoo_series to a binary .RData file
> save(zoo_series, file='zoo_series.RData')
```

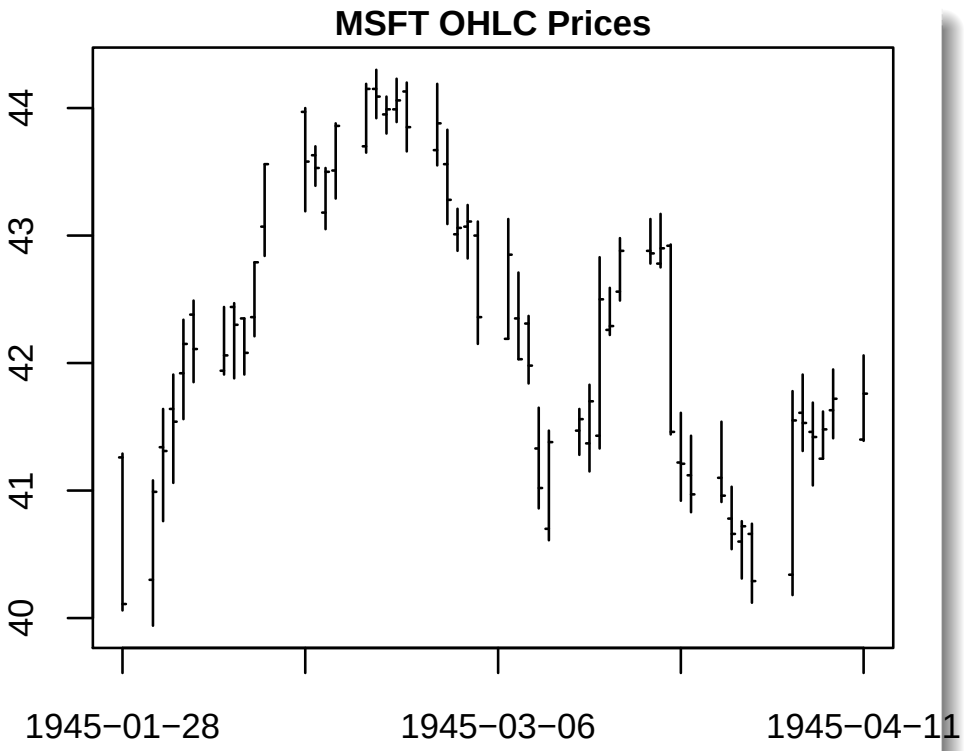
Plotting *OHLC* Time Series Using Package *tseries*

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,

```
> # get start and end dates
> in_dex <- time(ts_stx_adj)
> e_nd <- in_dex[length(in_dex)]
> st_art <- round((4*e_nd + in_dex[1])/5)
> # plot using plotOHLC
> plotOHLC(window(ts_stx_adj,
+           start=st_art,
+           end=e_nd)[, 1:4],
+    xlab="", ylab="")
> title(main="MSFT OHLC Prices")
```

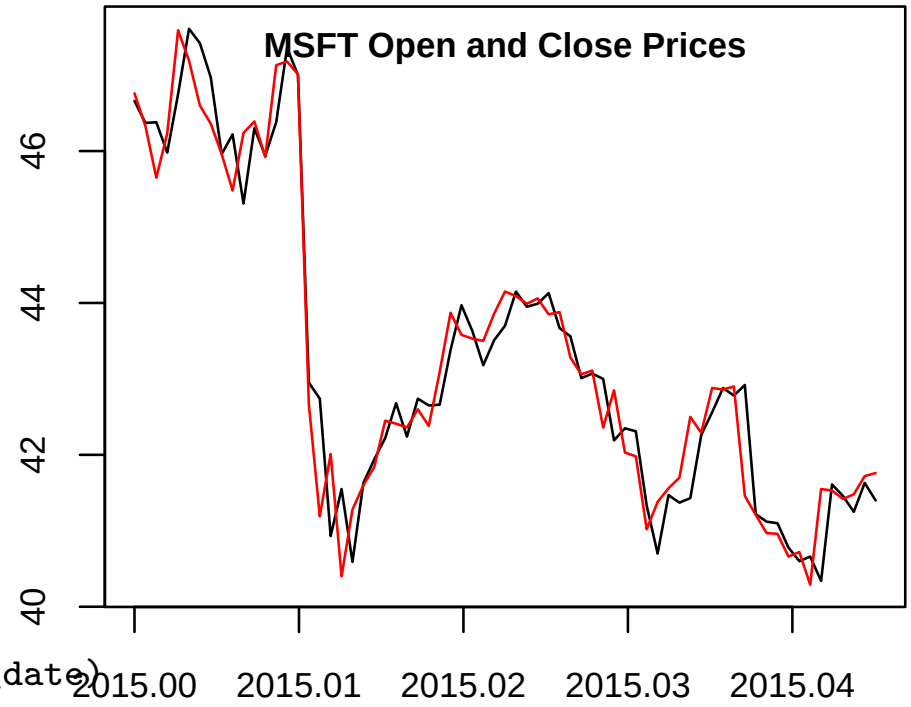


Plotting Two Time Series Using *tseries*

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

A *ts* time series can be created from a *zoo* 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,




```
> library(lubridate)  # load lubridate
> # get start and end dates of zoo_series
> start_date <- decimal_date(start(zoo_stx))
> end_date <- decimal_date(end(zoo_stx))
> # calculate frequency of zoo_stx
> fre_quency <- length(zoo_stx)/(end_date-start_date)
> # extract data from zoo_stx
> da_ta <- coredata(
+   window(zoo_stx, start=as.Date("2015-01-01"),
+    end=end(zoo_stx)))
> # create ts object using ts()
> ts_stx <- ts(data=da_ta, start=decimal_date(as.Date("2015-01-01")),
+           frequency=fre_quency)
> seqplot.ts(x=ts_stx[, 1], y=ts_stx[, 4], xlab="", ylab="")
> title(main="MSFT Open and Close Prices", line=-1)
```

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),

```
> # calculate maximum drawdown
> maxdrawdown(zoo_stx_adj[, "AdjClose"])
> max_drawd <- maxdrawdown(zoo_stx_adj[, "AdjClo
> index(zoo_stx_adj)[max_drawd$from]
> index(zoo_stx_adj)[max_drawd$to]
> # calculate Sharpe ratio
> sharpe(zoo_stx_adj[, "AdjClose"])
> # calculate Sterling ratio
> sterling(as.numeric(zoo_stx_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,

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

Fitting Time Series Models Using *tseries*

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

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

```
> zoo_stx <- suppressWarnings(  # load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(zoo_stx)
> dim(zoo_stx)
> tail(zoo_stx, 4)
>
> # calculate Sharpe ratio
> sharpe(zoo_stx[, "Close"], r=0.01)
> # add title
> plot(zoo_stx[, "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,

```
> zoo_stx <- suppressWarnings(  # load MSFT data
+   get.hist.quote(instrument="MSFT",
+             start=Sys.Date()-365,
+             end=Sys.Date(),
+             origin="1970-01-01")
+ )  # end suppressWarnings
> class(zoo_stx)
> dim(zoo_stx)
> tail(zoo_stx, 4)
>
> # calculate Sharpe ratio
> sharpe(zoo_stx[, "Close"], r=0.01)
> # add title
> plot(zoo_stx[, "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 uses time series objects 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()`, `Ad()`, `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")
```

ETF Dataset

```
> # ETF symbols for asset allocation
> sym_bols <- c("VTI", "VEU", "IEF", "VNQ",
+   "DBC", "VXX", "XLY", "XLP", "XLE", "XLF",
+   "XLV", "XLI", "XLB", "XLK", "XLU", "VYM",
+   "IVW", "IWB", "IWD", "IWF")
> # read etf database into data frame
> etf_list <- read.csv(
+   file='C:/Develop/R/lecture_slides/data/etf_list.csv',
+         stringsAsFactors=FALSE)
> rownames(etf_list) <- etf_list$Symbol
> # subset etf_list only those ETF's in sym_bols
> etf_list <- etf_list[sym_bols, ]
> # shorten names
> etf_names <- sapply(etf_list$Name,
+               function(name) {
+   name_split <- strsplit(name, split=" ")[[1]]
+   name_split <-
+     name_split[c(-1, -length(name_split))]
+   name_match <- match("Select", name_split)
+   if (!is.na(name_match))
+     name_split <- name_split[-name_match]
+   paste(name_split, collapse=" ")
+ })  # end sapply
> etf_list$Name <- etf_names
> etf_list["IEF", "Name"] <- "Treasury Bond Fund"
> etf_list["XLY", "Name"] <- "Consumer Discr. Sector Fund"
```

Symbol	Name	Fund.Type
VTI	Total Stock Market	US Equity ETF
VEU	FTSE All World Ex US	Global Equity ETF
IEF	Treasury Bond Fund	US Fixed Income ETF
VNQ	REIT ETF - DNQ	US Equity ETF
DBC	DB Commodity Index Trac	Commodity Based ETF
VXX	S&P500 VIX Futures	Commodity Based ETN
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

ETFs with names *X** represent industry sector funds,

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

IWB is the Russell 1000 small-cap fund,

VXX is the VIX volatility fund,

Downloading Time Series Data Using Package *quantmod*

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

`getSymbols()` creates objects in the specified *environment* from the input strings (names),

It then assigns the data to those objects, without returning them as a function value, as a *side effect*,

By default, `getSymbols()` downloads for each symbol the daily *OHLC* prices and trading volume (Open, High, Low, Close, Adjusted, Volume),

The method `getSymbols.yahoo` accepts arguments `"from"` and `"to"` which specify the date range for the data,

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

```
> library(quantmod)  # load package quantmod
> env_etf <- new.env()  # new environment for da
> # download data for sym_bols into env_etf
> getSymbols(sym_bols, env=env_etf, adjust=TRUE,
+     from="2007-01-03")
```

```
> ls(env_etf)  # list files in env_etf
> # get class of object in env_etf
> class(get(x=sym_bols[1], envir=env_etf))
> # another way
> class(env_etf$VTI)
> colnames(env_etf$VTI)
> head(env_etf$VTI, 3)
> # get class of all objects in env_etf
> eapply(env_etf, class)
> # get class of all objects in R workspace
> lapply(ls(), function(ob_ject) class(get(ob_je
```

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 `assign()` function 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,

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(env_etf$VTI)
> # adjust single OHLC object using its name
> env_etf$VTI <- adjustOHLC(env_etf$VTI,
+                           use.Adjusted=TRUE)
>
> # adjust OHLC object using string as name
> assign(sym_bols[1], adjustOHLC(
+     get(x=sym_bols[1], envir=env_etf),
+     use.Adjusted=TRUE),
+   envir=env_etf)
>
> # adjust objects in environment using vector of strings
> for (sym_bol in sym_bols) {
+   assign(sym_bol,
+    adjustOHLC(get(sym_bol, envir=env_etf),
+               use.Adjusted=TRUE),
+    envir=env_etf)
+ }  # end for
```

Extracting Prices Using Package *quantmod*

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

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

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

The function `do.call()` passes the list elements individually, instead of passing the whole list as one argument,

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,

The extractor (accessor) functions `Ad()`, `Vo()`, etc., extract columns from *OHLC* data,

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

```
> # extract and merge all data, subset by symbol
> price_s <- do.call(merge,
+   as.list(env_etf)[sym_bols])
> # or
> price_s <- rutils::do_call(cbind,
+   as.list(env_etf)[sym_bols])
> # extract and merge adjusted prices, subset by
> price_s <- rutils::do_call(cbind,
+   lapply(as.list(env_etf)[sym_bols], Ad))
> # same, but works only for OHLC series
> price_s <- rutils::do_call(cbind,
+   eapply(env_etf, Ad)[sym_bols])
> # drop ".Adjusted" from colnames
> colnames(price_s) <-
+   sapply(colnames(price_s),
+     function(col_name)
+ strsplit(col_name, split="[.]")[[1]])[1, ]
> tail(price_s[, 1:2], 3)
> # which objects in global environment are clas
> unlist(eapply(globalenv(), is.xts))
> # save xts to csv file
> write.zoo(price_s,
+   file='etf_series.csv', sep=",")
> # copy price_s into env_etf and save to .RData
> assign("price_s", price_s, envir=env_etf)
> save(env_etf, file='etf_data.RData')
```

Calculating Returns from Adjusted Prices

```
> # calculate returns from adjusted prices
> re_turns <- lapply(env_etf$price_s, function(x
+ # dailyReturn returns single xts with bad colu
+   daily_return <- dailyReturn(x_ts)
+   colnames(daily_return) <- names(x_ts)
+   daily_return
+ })  # end lapply
>
> # "re_turns" is a list of xts
> class(re_turns)
> class(re_turns[[1]])
>
> # flatten list of xts into a single xts
> re_turns <- do.call(merge, re_turns)
```

```
> class(re_turns)
> dim(re_turns)
> head(re_turns[, 1:3])
> # copy re_turns into env_etf and save to .RDat
> assign("re_turns", re_turns, envir=env_etf)
> save(env_etf, file='etf_data.RData')
```

Managing Data Inside Environments

The function `as.environment()` coerces objects (lists) 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,

```
> start_date <- "2012-05-10"; end_date <- "2013-11-20"
> # subset all objects in environment and return as enviro
> new_env <- as.environment(eapply(env_etf, "[",
+             paste(start_date, end_date, sep="/")))
> # subset only sym_bols in environment and return as envi
> new_env <- as.environment(
+   lapply(as.list(env_etf)[sym_bols], "[",
+    paste(start_date, end_date, sep="/")))
> # extract and merge adjusted prices and return to enviro
> assign("price_s", do.call(merge,
+          lapply(ls(env_etf), function(sym_bol) {
+            x_ts <- Ad(get(sym_bol, env_etf))
+            colnames(x_ts) <- sym_bol
+            x_ts
+          })), envir=new_env)
> # get sizes of OHLC xts series in env_etf
> sapply(mget(sym_bols, envir=env_etf), object.size)
> # extract and merge adjusted prices and return to enviro
> col_name <- function(x_ts)
+   strsplit(colnames(x_ts), split="[.]")[[1]][1]
> assign("price_s", do.call(merge,
+          lapply(mget(env_etf$sym_bols, envir=env_etf),
+                 function(x_ts) {
+                   x_ts <- Ad(x_ts)
+                   colnames(x_ts) <- col_name(x_ts)
+                   x_ts
+          })), envir=new_env)
```

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,

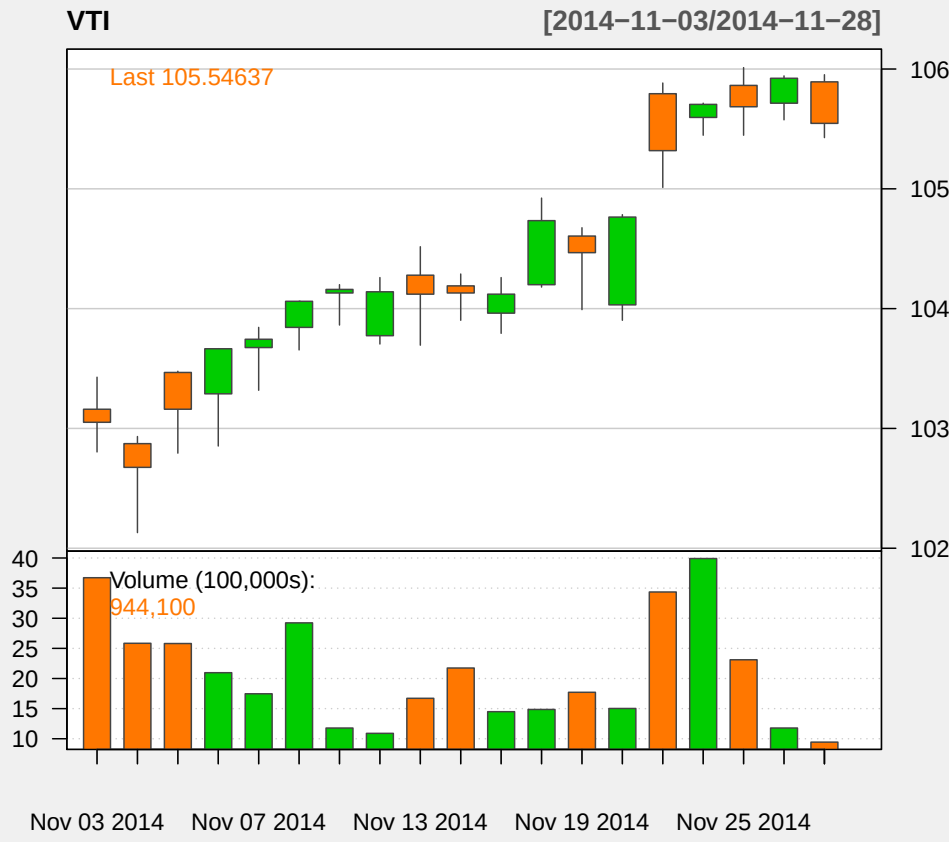
The argument `"type"` determines the type of plot (candlesticks, bars, or lines),

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(env_etf$VTI["2014-11"],
+       name="VTI",
+       theme=chartTheme("white"))
> # plot OHLC bar chart with volume
> chartSeries(env_etf$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,

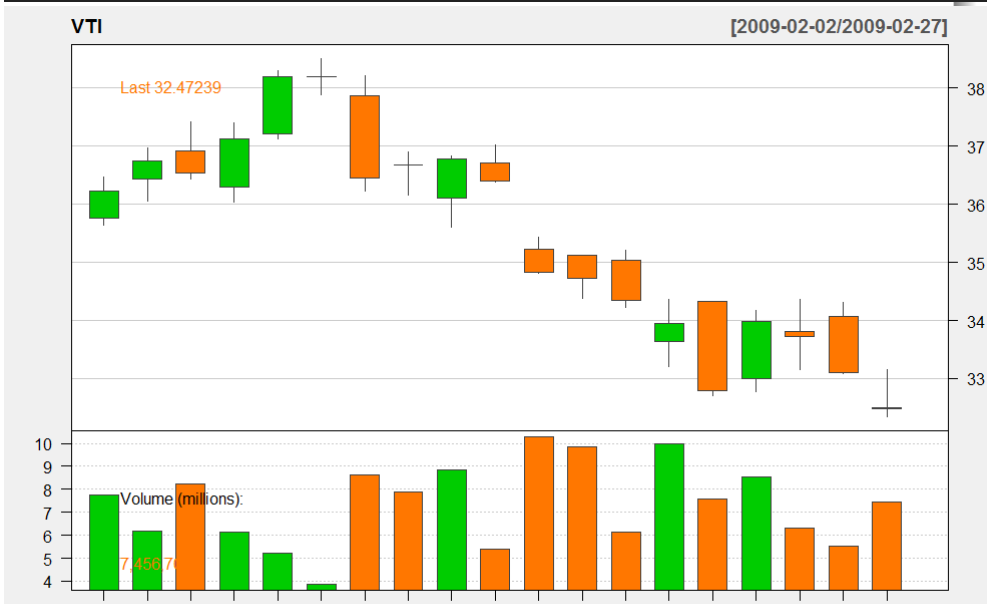
The color of the box signifies whether the *Close* price was higher or lower than the *Open*,

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(env_etf$VTI["2008-11/2009-04"],
+       name="VTI")
> # redraw plot only for Feb-2009, with white th
> reChart(subset="2009-02",
+   theme=chartTheme("white"))
```



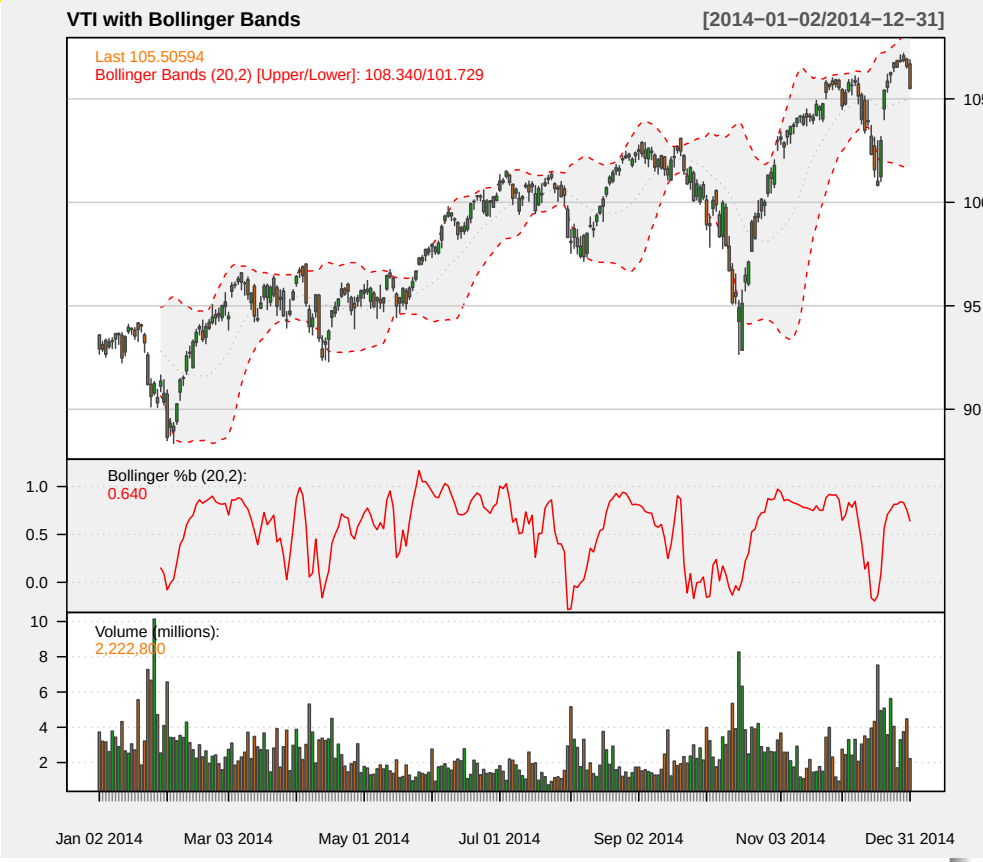
Plotting Technical Indicators Using chartSeries()

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

```
> oh_lc <- rutils::env_etf$VTI["2009-02/2009-03"]
> VTI_adj <- Ad(oh_lc); VTI_vol <- Vo(oh_lc)
> # calculate volume-weighted average price
> VTI_vwap <- TTR::VWAP(price=VTI_adj,
+ volume=VTI_vol, n=10)
> # plot OHLC candlechart with volume
> chartSeries(oh_lc, name="VTI plus VWAP",
+       theme=chartTheme("white"))
> # add VWAP to main plot
> addTA(ta=VTI_vwap, on=1, col='red')
> # add price minus VWAP in extra panel
> addTA(ta=(VTI_adj-VTI_vwap), 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 periods 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(oh_lc, name="VTI plus VWAP shaded"
+       theme=chartTheme("white"))
> # add VWAP to main plot
> addTA(ta=VTI_vwap, on=1, col='red')
> # add price minus VWAP in extra panel
> addTA(ta=(VTI_adj-VTI_vwap), col='red')
> # add background shading of areas
> addTA((VTI_adj-VTI_vwap) > 0, on=-1,
+ col="lightgreen", border="lightgreen")
> addTA((VTI_adj-VTI_vwap) < 0, on=-1,
+ col="lightgrey", border="lightgrey")
> # add vertical and horizontal lines at VTI_vwa
> addLines(v=which.min(VTI_vwap), col='red')
> addLines(h=min(VTI_vwap), 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=oh_lc, # volume in extra panel
+        TA="add_Vo(); add_TA(VTI_vwap, on=1)",
+        name="VTI plus VWAP shaded")
> # add price minus VWAP in extra panel
> add_TA(VTI_adj-VTI_vwap, col='red')
> # add background shading of areas
> add_TA((VTI_adj-VTI_vwap) > 0, on=-1,
+ col="lightgreen", border="lightgreen")
> add_TA((VTI_adj-VTI_vwap) < 0, on=-1,
+ col="lightgrey", border="lightgrey")
> # add vertical and horizontal lines
> abline(v=which.min(VTI_vwap), col='red')
> abline(h=min(VTI_vwap), 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
> ch_ob <- chart_Series(x=oh_lc, plot=FALSE)
> class(ch_ob)
> ls(ch_ob)
> class(ch_ob$get_ylim)
> class(ch_ob$set_ylim)
> # ls(ch_ob$Env)
> class(ch_ob$Env$actions)
> plot_theme <- chart_theme()
> class(plot_theme)
> ls(plot_theme)
```

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*,

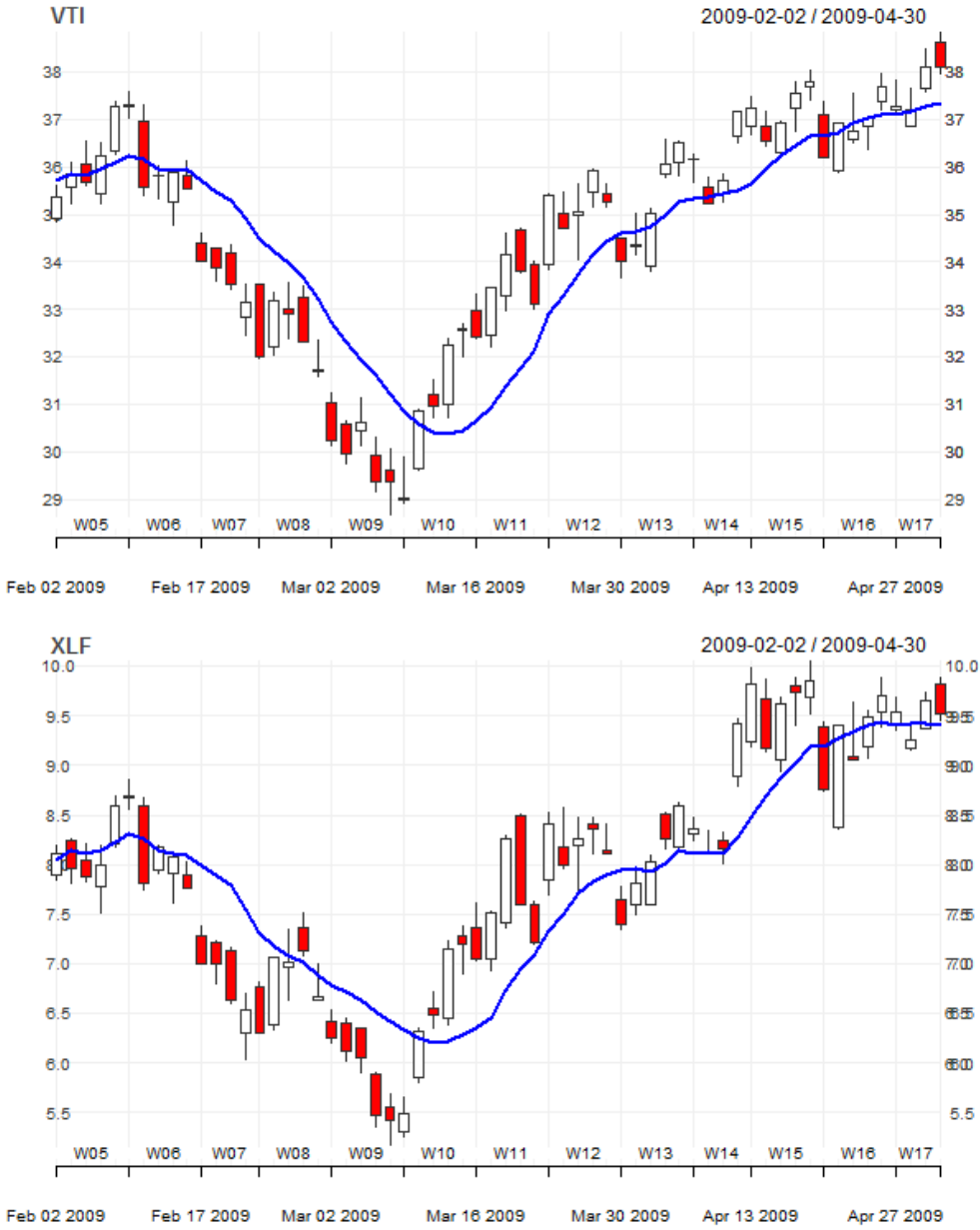
```
> oh_lc <- rutils::env_etf$VTI["2010-04/2010-05"]
> # extract, modify theme, format tick marks "%b %d"
> plot_theme <- chart_theme()
> plot_theme$format.labels <- "%b %d"
> # create plot object
> ch_ob <- chart_Series(x=oh_lc,
+                       theme=plot_theme, plot=FALSE)
> # extract ylim using accessor function
> y_lim <- ch_ob$get_ylim()
> y_lim[[2]] <- structure(
+   range(Ad(oh_lc)) + c(-1, 1),
+   fixed=TRUE)
> # modify plot object to reduce y-axis range
> ch_ob$set_ylim(y_lim)  # use setter function
> # render the plot
> plot(ch_ob)
```

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 averag
> VTI_vwap <-
+   TTR::VWAP(price=Ad(rutils::env_etf$VTI),
+       volume=Vo(rutils::env_etf$VTI), n=10)
> XLF_vwap <-
+   TTR::VWAP(price=Ad(rutils::env_etf$XLF),
+       volume=Vo(rutils::env_etf$XLF), n=10)
> # open graphics device, and define
> # plot area with two horizontal panels
> x11(); par(mfrow=c(2, 1))
> ch_ob <- chart_Series(  # plot in top panel
+   x=env_etf$VTI["2009-02/2009-04"],
+   name="VTI", plot=FALSE)
> add_TA(VTI_vwap["2009-02/2009-04"],
+  lwd=2, on=1, col='blue')
> ch_ob <- chart_Series(  # plot in bottom panel
+   x=env_etf$XLF["2009-02/2009-04"],
+   name="XLF", plot=FALSE)
> add_TA(XLF_vwap["2009-02/2009-04"],
+  lwd=2, on=1, col='blue')
```



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,

```
> library(dygraphs)
> # calculate volume-weighted average price
> oh_lc <- rutils::env_etf$VTI
> VTI_vwap <- TTR::VWAP(price=quantmod::Ad(oh_lc
+     volume=quantmod::Vo(oh_lc), n=20)
> # add VWAP to OHLC  data
> oh_lc <- cbind(oh_lc[, c(1:3, 6)],
+          VTI_vwap)["2009-02/2009-04"]
> # create dygraphs object
> dy_graph <- dygraphs::dygraph(oh_lc)
> # convert dygraphs object to candlestick plot
> dy_graph <- dygraphs::dyCandlestick(dy_graph)
> # render candlestick plot
> dy_graph
> # candlestick plot using pipes syntax
> dygraphs::dygraph(oh_lc) %>% dyCandlestick()
> # candlestick plot without using pipes syntax
> dygraphs::dyCandlestick(dygraphs::dygraph(oh_lc))
```

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

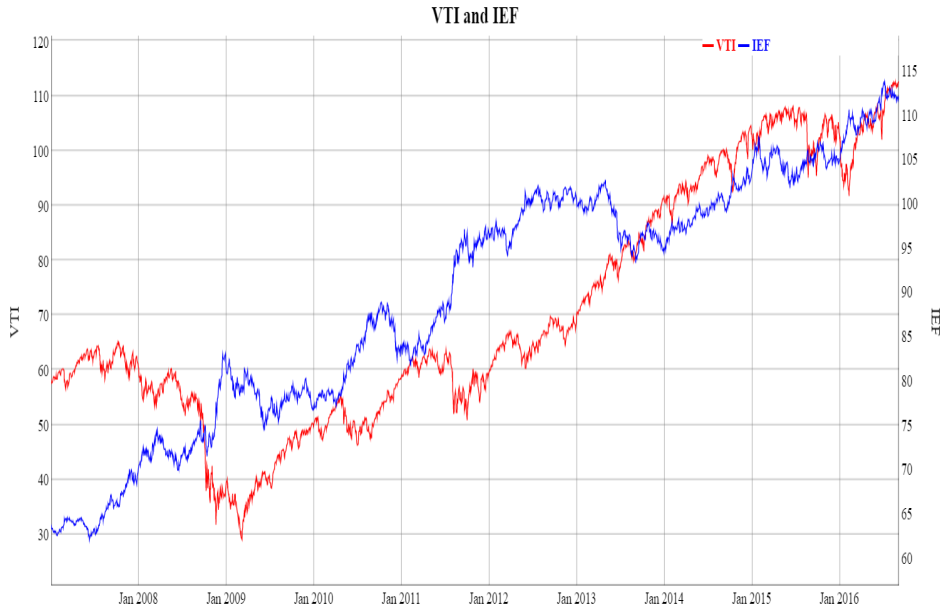
The function dyShading() adds shading to a *dygraphs* plot object,


```
> # create candlestick plot with background shad
> in_dex <- index(oh_lc)
> in_dic <-
+   rutils::diff_xts(oh_lc[, 4] > oh_lc[, "VWAP"
> in_dic <- rbind(cbind(which(in_dic==1), 1),
+   cbind(which(in_dic==(-1)), -1))
> in_dic <- in_dic[order(in_dic[, 1]), ]
> in_dic <- rbind(c(1, -in_dic[1, 2]), in_dic,
+   c(NROW(oh_lc), -in_dic[NROW(in_dic), 2]))
> in_dic <-
+   data.frame(in_dex[in_dic[, 1]], in_dic[, 2])
> # create dygraphs object
> dy_graph <- dygraphs::dygraph(oh_lc) %>%
+   dyCandlestick()
> # add shading
> for (i in 1:(NROW(in_dic)-1)) {
+   if (in_dic[i, 2] == 1)
+     dy_graph <- dy_graph %>% dyShading(from=in_dic[i, 1], to=in_dic[i+1, 1], color="lightgreen
+   else
+     dy_graph <- dy_graph %>% dyShading(from=in_dic[i, 1], to=in_dic[i+1, 1], color="antiquewhi
+ }  # end for
> # render plot
> dy_graph
```

dygraphs Plots With Two "y" Axes

The function `dyAxis()` from package *dygraphs* adds 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
> price_s <- cbind(Ad(rutils::env_etf$VTI),
+                  Ad(rutils::env_etf$IEF))
> col_names <- rutils::get_name(colnames(price_s
> colnames(price_s) <- col_names
>
> # dygraphs plot with two y-axes
> library(dygraphs)
> dygraphs::dygraph(price_s, main=paste(col_name
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(col_names[2], axis="y2", col=c("red", "blue"))
```

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,

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,

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

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 a `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
> sp_500 <- getURL(
+   "https://en.wikipedia.org/wiki/List_of_S%26P
> # extract tables from the text data
> sp_500 <- readHTMLTable(sp_500,
+                stringsAsFactors=FALSE)
> str(sp_500)
> # extract colnames of data frames
> lapply(sp_500, colnames)
> # extract S&P500 constituents
> sp_500 <- sp_500[[1]]
> head(sp_500)
> # create valid R names from symbols containing
> sp_500$names <- gsub("-", "_", sp_500$Ticker)
> sp_500$names <- gsub("[.]", "_", sp_500$names)
> # write data frame of S&P500 constituents to C
> write.csv(sp_500,
+   file="C:/Develop/R/lecture_slides/data/sp500
+   row.names=FALSE)
```

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

Before time series data for S&P500 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(HighFreq)  # load package HighFreq
> # load data frame of S&P500 constituents from CSV file
> sp_500 <- read.csv(file="C:/Develop/R/lecture_slides/dat
+      stringsAsFactors=FALSE)
> # register symbols corresponding to R names
> for (in_dex in 1:NROW(sp_500)) {
+   cat("processing: ", sp_500$Ticker[in_dex], "\n")
+   setSymbolLookup(structure(
+     list(list(name=sp_500$Ticker[in_dex])),
+     names=sp_500$names[in_dex]))
+ }  # end for
> env_sp500 <- new.env()  # new environment for data
> # remove all files (if necessary)
> rm(list=ls(env_sp500), envir=env_sp500)
> # download data and copy it into environment
> rutils::get_symbols(sp_500$names,
+    env_out=env_sp500, start_date="1990-01-01")
> # or download in loop
> for (sym_bol in sp_500$names) {
+   cat("processing: ", sym_bol, "\n")
+   rutils::get_symbols(sym_bol,
+    env_out=env_sp500, start_date="1990-01-01")
+ }  # end for
> save(env_sp500, file="C:/Develop/R/lecture_slides/data/s
> chart_Series(x=env_sp500$BRK_B["2016/"], TA="add_Vo()",
+        name="BRK-B stock")
```

Downloading *FRED* Time Series Data

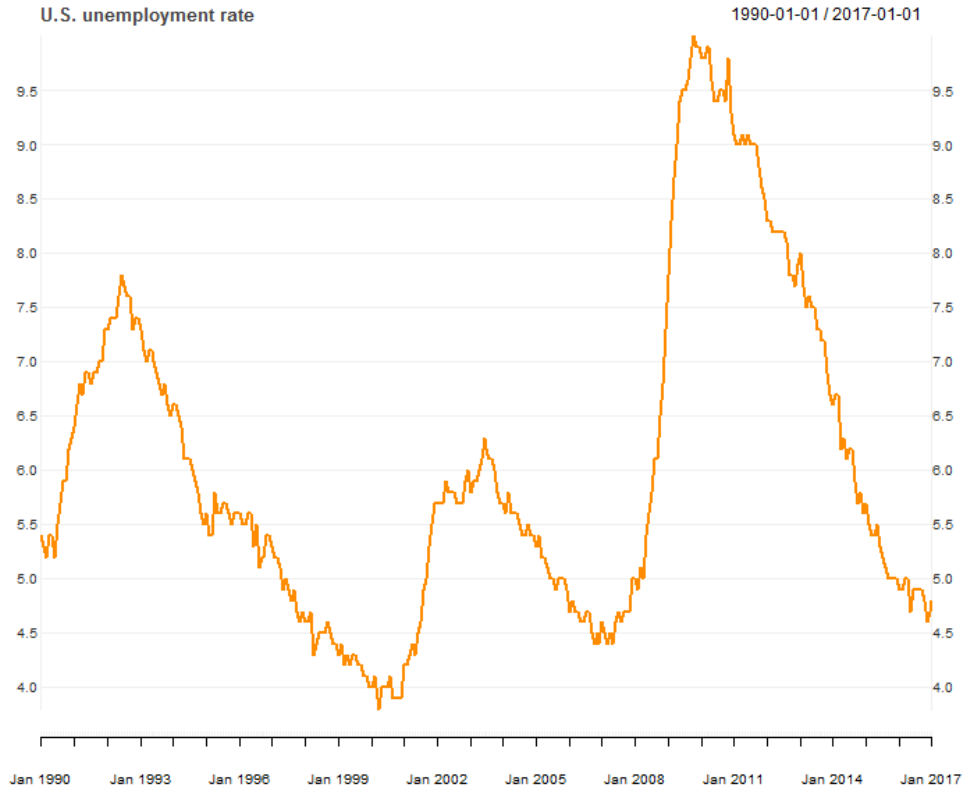
FRED is a database of economic time series maintained by the Federal Reserve Bank of St. Louis:

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
> unemp_rate <- getSymbols("UNRATE",
+              auto.assign=FALSE,
+              src="FRED")
> # plot U.S. unemployment rate data
> chart_Series(unemp_rate["1990/"],
+       name="U.S. unemployment rate")
```

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 offers 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 offers stock prices, stock fundamentals, financial ratios, 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(quantmod)  # load package quantmod
> # download EOD AAPL prices from WIKI free data
> price_s <- Quandl(code="WIKI/AAPL",
+             type="xts", start_date="1990-01-01
> x11(width=14, height=7)
> chart_Series(price_s["2016", 1:4],
+     name="AAPL OHLC prices")
> # add trade volume in extra panel
> add_TA(price_s["2016", 5])
> # download euro currency rates
> price_s <- Quandl(code="BNP/USDEUR",
+     start_date="2013-01-01",
+     end_date="2013-12-01", type="xts")
> # download multiple time series
> price_s <- Quandl(code=c("NSE/OIL", "WIKI/AAPL
+     start_date="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
> price_s <- Quandl(code="PE/AAPL_HURST",
+     start_date="2013-01-01", type="xts")
> chart_Series(price_s["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-lists

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
> sp_500 <- read.csv(
+   file="C:/Develop/R/lecture_slides/data/sp500
+   stringsAsFactors=FALSE)
> # replace "-" with "_" in symbols
> sp_500$free_code <-
+   gsub("-", "_", sp_500$free_code)
> head(sp_500)
> # vector of symbols in sp_500 frame
> tick_ers <- gsub("-", "_", sp_500$ticker)
> # or
> tick_ers <- matrix(unlist(
+   strsplit(sp_500$free_code, split="/"),
+   use.names=FALSE), ncol=2, byrow=TRUE)[, 2]
> # or
> tick_ers <- do_call_rbind(
+   strsplit(sp_500$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),

```
> env_sp500 <- new.env()  # new environment for data
> # remove all files (if necessary)
> rm(list=ls(env_sp500), envir=env_sp500)
> # Boolean vector of symbols already downloaded
> down_loaded <- tick_ers %in% ls(env_sp500)
> # download data and copy it into environment
> for (tick_er in tick_ers[!down_loaded]) {
+   cat("processing: ", tick_er, "\n")
+   da_ta <- Quandl(code=paste0("WIKI/", tick_er),
+             start_date="1990-01-01",
+             type="xts")[, -(1:7)]
+   colnames(da_ta) <- paste(tick_er,
+     c("Open", "High", "Low", "Close", "Volume"), sep=".
+   assign(tick_er, da_ta, envir=env_sp500)
+ }  # end for
> save(env_sp500, file="C:/Develop/R/lecture_slides/data/s
> chart_Series(x=env_sp500$XOM["2016/"], TA="add_Vo()",
+        name="XOM stock")
```

Homework Assignment

Required

- Read all the lecture slides in `FRE7241_Lecture_1.pdf`, and run all the code in `FRE7241_Lecture_1.R`,
- Download from Google Drive and study:
  functions.pdf: pages 29-40,
  statistics.pdf: pages 1-14,
  https://drive.google.com/drive/u/0/folders/0Bxzva1I0t63vVGEtaXNIY1JMa00

Recommended

- Read the documentation for packages `rutils.pdf` and `HighFreq.pdf`,
- Download and study additional materials from Google Drive:
  https://drive.google.com/drive/u/0/folders/0Bxzva1I0t63vVGEtaXNIY1JMa00