Data Management

FRE6871 & FRE7241, Spring 2025

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

March 15, 2025

Writing Text Strings

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

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

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

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

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

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

Displaying Numeric Data

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

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

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

Reading Text from Files

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Writing and Reading *Data Frames* From .csv Files

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

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

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

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

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

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

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

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

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

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

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

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

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

Reading Data From Very Large .csv Files

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

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

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

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

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

Writing and Reading Matrices From .csv Files

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

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

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

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

Writing and Reading Matrices (cont.)

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

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

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

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

Removing row names simplifies the writing and reading of matrices.

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

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

Reading Matrices Containing Bad Data

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

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

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

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

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

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

Writing and Reading Time Series From *Text* Files

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

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

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

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

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

Writing and Reading Time Series From .csv Files

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

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

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

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

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

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

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

The function `write.zoo()` writes *zoo* time series into `.csv` files, but it doesn't format the time at midnight properly.

The function `write.table()` writes *zoo* time series into `.csv` files, and it formats the time at midnight properly.

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

The functions `read.csv.zoo()` and `read.zoo()` read *zoo* time series from `.csv` files.

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

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

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

```
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Create zoo with POSIXct date-time index
> datev <- seq(from=as.POSIXct("2014-07-14"),
+         by="hour", length.out=100)
> zooseries <- zoo(rnorm(NROW(datev)), order.by=datev)
> head(zooseries, 3)
> # Write zoo series to CSV file using write.zoo()
> write.zoo(zooseries, file="zooseries.csv", sep=",", col.names=TRUE
> # Read from CSV file using read.csv.zoo() - doesn't work
> zooread <- read.csv.zoo(file="zooseries.csv", header=FALSE,
+   format="%Y-%m-%d %H:%M:%S", tz="America/New_York")
> # Read from CSV file using read.zoo() - error
> zooread <- read.zoo(file="zooseries.csv", header=FALSE,
+   sep=",", FUN=as.POSIXct, format="%Y-%m-%d %H:%M:%S")
> # Write zoo series to CSV file using write.table()
> write.table(zooseries, file="zooseries.csv", sep=",",
+       row.names=TRUE, col.names=FALSE)
> # Read from CSV file using read.zoo() with format argument
> zooread <- read.zoo(file="zooseries.csv", header=FALSE,
+   sep=",", FUN=as.POSIXct, format="%Y-%m-%d %H:%M:%S")
> all.equal(zooseries, zooread) # Works
> # Coerce zoo series into data frame with custom date format
> dframe <- as.data.frame(zooseries)
> rownames(dframe) <- format(index(zooseries), format="%m-%d-%Y %H:%
> # Write zoo series to csv file using write.table
> write.table(dframe, file="zooseries.csv", sep=",",
+       row.names=TRUE, col.names=FALSE)
> # Read from CSV file using read.zoo()
> zooread <- read.zoo(file="zooseries.csv", header=FALSE, sep=",",
+   FUN=as.POSIXct, format="%m-%d-%Y %H:%M:%S")
> all.equal(zooseries, zooread) # Works
> # Or using read.csv.zoo()
> zooread <- read.csv.zoo(file="zooseries.csv", header=FALSE,
+   format="%m-%d-%Y %H:%M:%S", tz="America/New_York")
> head(zooread, 3)
> all.equal(zooseries, zooread, check.attributes=FALSE) # Works
```

Reading Time Series With `Numeric` *Date-time* Index

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

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

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

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

Passing Arguments to the save() Function

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

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

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

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

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

```
> var1 <- 1; var2 <- 2
> ls()  # List all objects
> ls()[1]  # List first object
> args(save)  # List arguments of save function
> # Save "var1" to a binary file using string argument
> save("var1", file="my_data.RData")
> # Save "var1" to a binary file using object name
> save(var1, file="my_data.RData")
> # Save multiple objects
> save(var1, var2, file="my_data.RData")
> # Save first object in list by passing to "..." argument
> # ls()[1] is not evaluated
> save(ls()[1], file="my_data.RData")
> # Save first object in list by passing to "list" argument
> save(list=ls()[1], file="my_data.RData")
> # Save whole list by passing it to the "list" argument
> save(list=ls(), file="my_data.RData")
```

Writing and Reading Lists of Objects

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

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

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

Saving Output of R to a File

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

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

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

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

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

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

Package *data.table* for High Performance Data Management

The package *data.table* is designed for high performance data management.

The package *data.table* implements *data table* objects, which are a special type of *data frame*, and an extension of the *data frame* class.

Data tables are faster and more convenient to work with than *data frames*.

data.table functions are optimized for high performance (speed), because they are written in C++ and they perform operations by reference (in place), without copying data in memory.

Some of the attractive features of package *data.table* are:

- Syntax is analogous to SQL,
- Very fast writing and reading from files,
- Very fast sorting and merging operations,
- Subsetting using multiple logical clauses,
- Columns of type `character` are never converted to factors,

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

The package *data.table* has extensive documentation:
https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html
https://github.com/Rdatatable/data.table/wiki

Data Table Objects

Data table objects are a special type of *data frame*, and are derived from the class data.frame.

Data table objects resemble databases, with columns of different types of data, and rows of records containing individual observations.

The function data.table::data.table() creates a *data table* object.

Data table columns can be referenced directly by their names (without quotes), and their rows can be referenced without a following comma.

When a *data table* is printed (by typing its name) then only the top 5 and bottom 5 rows are displayed (unless getOption("datatable.print.nrows") is less than 100).

The operator .N returns the number of observations (rows) in the *data table*.

Data table computations are usually much faster than equivalent R computations, but not always.

```
> # Create a data table
> library(data.table)
> dtable <- data.table::data.table(
+   col1=sample(7), col2=sample(7), col3=sample(7))
> # Print dtable
> class(dtable); dtable
> # Column referenced without quotes
> dtable[, col2]
> # Row referenced without a following comma
> dtable[2]
> # Print option "datatable.print.nrows"
> getOption("datatable.print.nrows")
> options(datatable.print.nrows=10)
> getOption("datatable.print.nrows")
> # Number of rows in dtable
> NROW(dtable)
> # Or
> dtable[, NROW(col1)]
> # Or
> dtable[, .N]
> # microbenchmark speed of data.table syntax
> library(microbenchmark)
> summary(microbenchmark(
+   dt=dtable[, .N],
+   rcode=NROW(dtable),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Writing and Reading Data Using Package *data.table*

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

The function data.table::fread() reads from .csv files and returns a *data table* object of class data.table.

Data table objects are a special type of *data frame*, and are derived from the class data.frame.

The function data.table::fread() is over 6 times faster than read.csv()!

The function data.table::fwrite() writes to .csv files over 12 times faster than the function write.csv(), and 300 times faster than function cat()!

```
> # Read a data table from CSV file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> filen <- file.path(dirn, "weather_delays14.csv")
> dtable <- data.table::fread(filen)
> class(dtable); dim(dtable)
> dtable
> # fread() reads the same data as read.csv()
> all.equal(read.csv(filen),
+     setDF(data.table::fread(filen)))
> # fread() is much faster than read.csv()
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=read.csv(filen),
+   fread=setDF(data.table::fread(filen)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Write data table to file in different ways
> data.table::fwrite(dtable, file="dtable.csv")
> write.csv(dtable, file="dtable2.csv")
> cat(unlist(dtable), file="dtable3.csv")
> # microbenchmark speed of data.table::fwrite()
> summary(microbenchmark(
+   fwrite=data.table::fwrite(dtable, file="dtable.csv"),
+   write_csv=write.csv(dtable, file="dtable2.csv"),
+   cat=cat(unlist(dtable), file="dtable3.csv"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Subsetting *Data Table* Objects

The square braces (brackets) "[]" operator subsets (references) the rows and columns of *data tables*.

Data table rows can be subset without a following comma.

Data table columns can be referenced directly by their names (without quotes, as if they were variables), after a comma.

Multiple *data table* columns can be referenced by passing a list of names.

The brackets "[]" operator is a *data.table* function, and all the commands inside the brackets "[]" are executed using code from the package *data.table*.

The dot .() operator is equivalent to the list function list().

```
> # Select first five rows of dtable
> dtable[1:5]
> # Select rows with JFK flights
> jfkf <- dtable[origin=="JFK"]
> # Select rows JFK flights in June
> jfkf <- dtable[origin=="JFK" & month==6]
> # Select rows without JFK flights
> jfkf <- dtable[!(origin=="JFK")]
> # Select flights with carrier_delay
> dtable[carrier_delay > 0]
> # Select column of dtable and return a vector
> head(dtable[, origin])
> # Select column of dtable and return a dtable, not vector
> head(dtable[, list(origin)])
> head(dtable[, .(origin)])
> # Select two columns of dtable
> dtable[, list(origin, month)]
> dtable[, .(origin, month)]
> columnv <- c("origin", "month")
> dtable[, ..columnv]
> dtable[, month, origin]
> # Select two columns and rename them
> dtable[, .(orig=origin, mon=month)]
> # Select all columns except origin
> head(dtable[, !"origin"])
> head(dtable[, -"origin"])
```

Performing Computations on *Data Table* Columns

If the second argument in the brackets "[]" operator is a function of the columns, then the brackets return the result of the function's computations on those columns.

The second argument in the brackets "[]" can also be a list of functions, in which case the brackets return a vector of computations.

The brackets "[]" can evaluate most standard R functions, but they are executed using *data.table* code, which is usually much faster than the equivalent R functions.

The operator .N returns the number of observations (rows) in the *data table*.

```
> # Select flights with positive carrier_delay
> dtable[carrier_delay > 0]
> # Number of flights with carrier_delay
> dtable[, sum(carrier_delay > 0)]
> # Or standard R commands
> sum(dtable[, carrier_delay > 0])
> # microbenchmark speed of data.table syntax
> summary(microbenchmark(
+   dt=dtable[, sum(carrier_delay > 0)],
+   rcode=sum(dtable[, carrier_delay > 0]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Average carrier_delay
> dtable[, mean(carrier_delay)]
> # Average carrier_delay and aircraft_delay
> dtable[, .(carrier=mean(carrier_delay),
+      aircraft=mean(aircraft_delay))]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Number of flights from JFK
> dtable[origin=="JFK", NROW(aircraft_delay)]
> # Or
> dtable[origin=="JFK", .N]
> # In R
> sum(dtable[, origin]=="JFK")
```

Grouping *Data Table* Computations by Factor Columns

The *data table* brackets "[]" operator can accept three arguments: `[i, j, by]`

- `i`: the row index to select,
- `j`: a list of columns or functions on columns,
- `by`: the columns of factors to aggregate over.

The *data table* columns can be *aggregated* over categories (factors) defined by one or more columns passed to the `"by"` argument.

The `"keyby"` argument is similar to `"by"`, but it sorts the output according to the categories used to group by.

Multiple *data table* columns can be referenced by passing a list of names.

The dot `.()` operator is equivalent to the list function `list()`.

```
> # Number of flights from each airport
> dtable[, .N, by=origin]
> # Same, but add names to output
> dtable[, .(flights=.N), by=.(airport=origin)]
> # Number of AA flights from each airport
> dtable[carrier=="AA", .(flights=.N), by=.(airport=origin)]
> # Number of flights from each airport and airline
> dtable[, .(flights=.N), by=.(airport=origin, airline=carrier)]
> # Average aircraft_delay
> dtable[, mean(aircraft_delay)]
> # Average aircraft_delay from JFK
> dtable[origin=="JFK", mean(aircraft_delay)]
> # Average aircraft_delay from each airport
> dtable[, .(delay=mean(aircraft_delay)), by=.(airport=origin)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+      by=.(airport=origin, month=month)]
> # Average and max delays from each airport and month
> dtable[, .(mean_delay=mean(aircraft_delay), max_delay=max(aircraft
+      keyby=.(airport=origin, month=month)]
```

Sorting *Data Table* Rows by Columns

Standard R functions can be used inside the brackets "[]" operator.

The function order() calculates the permutation index, to sort a given vector into ascending order.

The function setorder() sorts the rows of a *data table* by reference (in place), without copying data in memory.

setorder() is over 10 times faster than order(), because it doesn't copy data in memory.

Several brackets "[]" operators can be chained together to perform several consecutive computations.

```
> # Sort ascending by origin, then descending by dest
> dtables <- dtable[order(origin, -dest)]
> dtables
> # Doesn't work outside dtable
> order(origin, -dest)
> # Sort dtable by reference
> setorder(dtable, origin, -dest)
> all.equal(dtable, dtables)
> # setorder() is much faster than order()
> summary(microbenchmark(
+   order=dtable[order(origin, -dest)],
+   setorder=setorder(dtable, origin, -dest),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # Average aircraft_delay by month
> dtables[, .(mean_delay=mean(aircraft_delay)),
+       by=.(month=month)]
> # Chained brackets to sort output by month
> dtables[, .(mean_delay=mean(aircraft_delay)),
+   by=.(month=month)][order(month)]
```

Subsetting, Computing, and Grouping *Data Table* Objects

The special symbol .SD selects a subset of a *data table*.

The symbol .SDcols specifies the columns to select by the symbol .SD.

Inside the brackets "[]" operator, the .SD symbol can be treated as a virtual *data table*, and standard R functions can be applied to it.

The "by" argument can be used to group the outputs produced by the functions applied to the .SD symbol.

If the symbol .SDcols is not defined, then the symbol .SD returns the remaining columns not passed to the "by" operator.

```
> # Select weather_delay and aircraft_delay in two different ways
> dtable[1:7, .SD,
+      .SDcols=c("weather_delay", "aircraft_delay")]
> dtable[1:7, .(weather_delay, aircraft_delay)]
> # Calculate mean of weather_delay and aircraft_delay
> dtable[, sapply(.SD, mean),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> sapply(dtable[, .SD,
+      .SDcols=c("weather_delay", "aircraft_delay")], mean)
> # Return origin and dest, then all other columns
> dtable[1:7, .SD, by=.(origin, dest)]
> # Return origin and dest, then weather_delay and aircraft_delay co
> dtable[1:7, .SD, by=.(origin, dest),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> # Return first two rows from each month
> dtable[, head(.SD, 2), by=.(month)]
> dtable[, head(.SD, 2), by=.(month),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> # Calculate mean of weather_delay and aircraft_delay, grouped by o
> dtable[, lapply(.SD, mean),
+      by=.(origin),
+      .SDcols=c("weather_delay", "aircraft_delay")]
> # Or simply
> dtable[, .(weather_delay=mean(weather_delay),
+          aircraft_delay=mean(aircraft_delay)),
+      by=.(origin)]
```

Modifying *Data Table* Objects by Reference

The special assignment operator ":=" allows modifying *data table* columns by reference (in place), without copying data in memory.

The computations on columns by reference can be *grouped* over categories defined by one or more columns passed to the "by" argument.

The computations are recycled to fit the size of each group.

The selected parts of columns can also be modified by reference, by combining the i and j arguments.

The special symbols .SD and .SDcols can be used to perform computations on several columns.

Modifying by reference is several times faster than standard R assignment.

```
> # Add tot_delay column
> dtable[, tot_delay := (carrier_delay + aircraft_delay)]
> head(dtable, 4)
> # Delete tot_delay column
> dtable[, tot_delay := NULL]
> # Add max_delay column grouped by origin and dest
> dtable[, max_delay := max(aircraft_delay), by=.(origin, dest)]
> dtable[, max_delay := NULL]
> # Add date and tot_delay columns
> dtable[, c("date", "tot_delay") :=
+       list(paste(month, day, year, sep="/"),
+            (carrier_delay + aircraft_delay))]
> # Modify select rows of tot_delay column
> dtable[month == 12, tot_delay := carrier_delay]
> dtable[, c("date", "tot_delay") := NULL]
> # Add several columns
> dtable[, c("max_carrier", "max_aircraft") := lapply(.SD, max),
+  by=.(origin, dest),
+  .SDcols=c("carrier_delay", "aircraft_delay")]
> # Remove columns
> dtable[, c("max_carrier", "max_aircraft") := NULL]
> # Modifying by reference is much faster than standard R
> summary(microbenchmark(
+   dt=dtable[, tot_delay := (carrier_delay + aircraft_delay)],
+   rcode=(dtable[, "tot_delay"] <- dtable[, "carrier_delay"] + dtab
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Adding *keys* to *Data Tables* for Fast Binary Search

The *key* of a *data table* is analogous to the row indices of a *data frame*, and it determines the ordering of its rows.

The function `data.table::setkey()` adds a *key* to a *data table*, and sorts the *data table* rows by reference according to the key.

`setkey()` creates the *key* from one or more columns of the *data frame*.

Subsetting rows using a *key* can be several times faster than standard R.

```
> # Add a key based on the "origin" column
> setkey(dtable, origin)
> haskey(dtable)
> key(dtable)
> # Select rows with LGA using the key
> dtable["LGA"]
> all.equal(dtable["LGA"], dtable[origin == "LGA"])
> # Select rows with LGA and JFK using the key
> dtable[c("LGA", "JFK")]
> # Add a key based on the "origin" and "dest" columns
> setkey(dtable, origin, dest)
> key(dtable)
> # Select rows with origin from JFK and MIA
> dtable[c("JFK", "MIA")]
> # Select rows with origin from JFK and dest to MIA
> dtable[.("JFK", "MIA")]
> all.equal(dtable[.("JFK", "MIA")],
+     dtable[origin == "JFK" & dest == "MIA"])
> # Selecting rows using a key is much faster than standard R
> summary(microbenchmark(
+   with_key=dtable[.("JFK", "MIA")],
+   standard_r=dtable[origin == "JFK" & dest == "MIA"],
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

draft: Subsetting and Grouping *Data Tables* Using *keys*

Subsetting and grouping *data tables* using *keys* can be several times faster than standard R.

The *key* of a *data table* is analogous to the row indices of a *data frame*, and it determines the ordering of its rows.

The function `data.table::setkey()` adds a *key* to a *data table*, and sorts the *data table* rows by reference according to the key.

`setkey()` creates the *key* from one or more columns of the *data frame*.

```
> # Select rows with dest to MIA
> dtable[.(unique(origin), "MIA")]
> # Select carrier_delay for all flights from JFK to MIA
> dtable[.("JFK", "MIA"), carrier_delay]
> dtable[.("JFK", "MIA"), .(carrier_delay)]
> dtable[.("JFK", "MIA"), .(carrier, carrier_delay)]
> # Calculate longest carrier_delay from JFK to MIA
> dtable[.("JFK", "MIA"), max(carrier_delay)]
> # Chain commands to sort the carrier_delay
> dtable[.("JFK", "MIA"), .(carrier, carrier_delay)][order(-carrier_
> dtable[.("JFK", "MIA"), .(carrier, carrier_delay)][carrier_delay>(
> # Calculate carrier with longest carrier_delay from JFK to MIA
> dtable[.("JFK", "MIA"), .(carrier, carrier_delay)][carrier_delay =
> # Calculate longest carrier_delay from JFK to every dest
> dtable["JFK", .(max_delay=max(carrier_delay)), keyby=.(dest)]
> # Calculate longest carrier_delay for every carrier, from JFK to e
> dtable["JFK", .(max_delay=max(carrier_delay)), keyby=.(dest, carri
> # Calculate carriers with longest carrier_delay from JFK to every
> # doesn't work
> dtable["JFK"][carrier_delay == max(carrier_delay), .(carrier, carr
> dtable["JFK",
+      lapply(.SD, function(x) x[max(carrier_delay)]),
+      by=.(dest),
+      .SDcols=c("dest", "carrier_delay")]
> # Set carrier_delay to longest carrier_delay from JFK to MIA
> dtable[.("JFK", "MIA", carrier_delay == max(carrier_delay)), carri
> # Show the modified row (record)
> dtable[.("JFK", "MIA")][carrier_delay == max(carrier_delay)]
```

draft: Splitting *Data Table* Objects

Data table columns can be referenced directly by their names (without quotes, as if they were variables), after a comma.

Data table rows can be subset without a following comma.

```
> # Select using multiple logical clauses
> jfkf <- dtable[origin=="JFK" & month==6]
> dim(dtable); dim(jfkf)
> # Select first five rows
> jfkf[1:5]
> # Sort data table by "origin" column in ascending order, then by '
> dtable <- dtable[order(origin, -dest)]
> # fsort() is much slower than sort() !
> datav <- runif(1e3)
> all.equal(sort(datav), data.table::fsort(datav))
> library(microbenchmark)
> summary(microbenchmark(
+   rcode=sort(datav),
+   dt=data.table::fsort(datav),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing *Data Table* Objects Into *Data Frames*

The functions `data.table::setDT()` and `data.table::setDF()` coerce *data frames* to *data tables*, and vice versa.

The *set* functions `data.table::set*()` perform their operations by reference (in place), without returning any values or copying data to a new memory location, which makes them very fast.

Data table objects can also be coerced into *data frames* using the function `as.data.frame()`, but it's much slower because it makes copies of data.

```
> # Create data frame and coerce it to data table
> dtable <- data.frame(col1=sample(7), col2=sample(7), col3=sample(7
> class(dtable); dtable
> data.table::setDT(dtable)
> class(dtable); dtable
> # Coerce dtable into data frame
> data.table::setDF(dtable)
> class(dtable); dtable
> # Or
> dtable <- data.table:::as.data.frame.data.table(dtable)
> # SetDF() is much faster than as.data.frame()
> summary(microbenchmark(
+   asdataframe=data.table:::as.data.frame.data.table(dtable),
+   setDF=data.table::setDF(dtable),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

draft: Coercing *Data Tables* Into xts Time Series

A *data table* can be coerced into an xts time series by first coercing it into a *data frame* and then into a *data table* using the function data.table::setDT().

But then the time index of the xts series is coerced into strings, not dates.

An xts time series can also be coerced directly into a *data table* using the function data.table::as.data.table().

```
> # Coerce xts to a data frame
> pricev <- rutils::etfenv$VTI
> class(pricev); head(pricev)
> pricev <- as.data.frame(pricev)
> class(pricev); head(pricev)
> # Coerce data frame to a data table
> data.table::setDT(pricev, keep.rownames=TRUE)
> class(pricev); head(pricev)
> # Dates are coerced to strings
> sapply(pricev, class)
> # Coerce xts directly to a data table
> dtable <- as.data.table(rutils::etfenv$VTI,
+   keep.rownames=TRUE)
> class(dtable); head(dtable)
> # Dates are not coerced to strings
> sapply(dtable, class)
> all.equal(pricev, dtable, check.attributes=FALSE)
```

Coercing xts Time Series Into *Data Tables*

An xts time series can be coerced into a *data table* by first coercing it into a *data frame* and then into a *data table* using the function data.table::setDT().

But then the time index of the xts series is coerced into strings, not dates.

An xts time series can also be coerced directly into a *data table* using the function data.table::as.data.table().

```
> # Coerce xts to a data frame
> pricev <- rutils::etfenv$VTI
> class(pricev); head(pricev)
> pricev <- as.data.frame(pricev)
> class(pricev); head(pricev)
> # Coerce data frame to a data table
> data.table::setDT(pricev, keep.rownames=TRUE)
> class(pricev); head(pricev)
> # Dates are coerced to strings
> sapply(pricev, class)
> # Coerce xts directly to a data table
> dtable <- as.data.table(rutils::etfenv$VTI,
+   keep.rownames=TRUE)
> class(dtable); head(dtable)
> # Dates are not coerced to strings
> sapply(dtable, class)
> all.equal(pricev, dtable, check.attributes=FALSE)
```

draft: The *IDateTime* date class

An xts time series can be coerced into a *data table* by first coercing it into a *data frame* and then into a *data table* using the function data.table::setDT().

But then the time index of the xts series is coerced into strings, not dates.

An xts time series can also be coerced directly into a *data table* using the function data.table::as.data.table().

```
> # Coerce xts to a data frame
> pricev <- rutils::etfenv$VTI
> class(pricev); head(pricev)
> pricev <- as.data.frame(pricev)
> class(pricev); head(pricev)
> # Coerce data frame to a data table
> data.table::setDT(pricev, keep.rownames=TRUE)
> class(pricev); head(pricev)
> # Dates are coerced to strings
> sapply(pricev, class)
> # Coerce xts directly to a data table
> dtable <- as.data.table(rutils::etfenv$VTI,
+   keep.rownames=TRUE)
> class(dtable); head(dtable)
> # Dates are not coerced to strings
> sapply(dtable, class)
> all.equal(pricev, dtable, check.attributes=FALSE)
```

Package *fst* for High Performance Data Management

The package *fst* provides functions for very fast writing and reading of *data frames* from *compressed binary files*.

The package *fst* writes to *compressed binary files* in the `fst` fast-storage format.

The package *fst* uses the `LZ4` and `ZSTD` compression algorithms, and utilizes multithreaded (parallel) processing on multiple CPU cores.

The package *fst* has extensive documentation:
http://www.fstpackage.org/

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

Writing and Reading Data Using Package *fst*

The package *fst* allows very fast writing and reading of *data frames* from *compressed binary files* in the fst fast-storage format.

The function `fst::write_fst()` writes to .fst files over 10 times faster than the function `write.csv()`, and 300 times faster than function `cat()` write to .csv files!

The function `fst::fread()` reads from .fst files over 10 times faster than the function `read.csv()` from .csv files!

```
> # Read a data frame from CSV file
> dirn <- "/Users/jerzy/Develop/lecture_slides/data/"
> filen <- file.path(dirn, "weather_delays14.csv")
> data.table::setDF(dframe)
> class(dframe); dim(dframe)
> # Write data frame to .fst file in different ways
> fst::write_fst(dframe, path="dframe.fst")
> write.csv(dframe, file="dframe2.csv")
> # microbenchmark speed of fst::write_fst()
> library(microbenchmark)
> summary(microbenchmark(
+   fst=fst::write_fst(dframe, path="dframe.csv"),
+   write_csv=write.csv(dframe, file="dframe2.csv"),
+   cat=cat(unlist(dframe), file="dframe3.csv"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
> # fst::read_fst() reads the same data as read.csv()
> all.equal(read.csv(filen),
+     fst::read_fst("dframe.fst"))
> # fst::read_fst() is 10 times faster than read.csv()
> summary(microbenchmark(
+   fst=fst::read_fst("dframe.fst"),
+   read_csv=read.csv(filen),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Random Access to Large Data Files

The package *fst* allows *random access* to very large *data frames* stored in compressed data files in the .fst format.

Data frames can be accessed *randomly* by loading only the selected rows and columns into memory, without fully loading the whole data frame.

function fst::fst() reads an .fst file and returns an *fst_table* reference object (pointer) to the data, without loading the whole data into memory.

The *fst_table* reference provides access to the data similar to a regular *data frame*, but it requires only a small amount of memory because the data isn't loaded into memory.

```
> # Coerce TAQ xts to a data frame
> library(HighFreq)
> taq <- HighFreq::SPY_TAQ
> taq <- as.data.frame(taq)
> class(taq)
> # Coerce data frame to a data table
> data.table::setDT(taq, keep.rownames=TRUE)
> class(taq); head(taq)
> # Get memory size of data table
> format(object.size(taq), units="MB")
> # Save data table to .fst file
> fst::write_fst(taq, path="/Users/jerzy/Develop/data/taq.fst")
> # Create reference to .fst file similar to a data frame
> refst <- fst::fst("/Users/jerzy/Develop/data/taq.fst")
> class(refst)
> # Memory size of reference to .fst is very small
> format(object.size(refst), units="MB")
> # Get sizes of all objects in workspace
> sort(sapply(mget(ls()), object.size))
> # Reference to .fst can be treated similar to a data table
> dim(taq); dim(refst)
> fst:::print.fst_table(refst)
> # Subset reference to .fst just like a data table
> refst[1e4:(1e4+5), ]
```

Downloading *ts* Time Series Using *tseries*

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

The default market data provider is *Yahoo* (provider="yahoo"), but *Yahoo* has stopped providing free market data.

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

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

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

Downloading *zoo* Time Series Using *tseries*

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

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

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

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

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

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

Adjusting *OHLC* Data

Stock prices experience jumps due to stock splits and dividends.

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

OHLC data can be adjusted for stock splits and dividends.

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

Downloading Data From *Oanda* Using *tseries*

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

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

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

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

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

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

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

Downloading Stock Prices Using *tseries*

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

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

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

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

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

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

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

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

The *ETF* Database

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

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

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

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

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

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

ETF Database for Investment Strategies

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

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

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

IWB is the Russell 1000 small-cap fund.

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

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

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

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

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

Exchange Traded Notes (*ETNs*)

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

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

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

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

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

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

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

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

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

Downloading ETF Prices Using Package *quantmod*

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

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

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

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

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

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

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

Inspecting ETF Prices in an Environment

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

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

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

Adjusting Stock Prices Using Package *quantmod*

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

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

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

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

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

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

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

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

Extracting Time Series from Environments

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

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

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

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

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

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

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

```
> library(rutils)  # Load package rutils
> # Define ETF symbols
> symbolv <- c("VTI", "VEU", "IEF", "VNQ")
> # Extract symbolv from rutils::etfenv
> pricev <- mget(symbolv, envir=rutils::etfenv)
> # pricev is a list of xts series
> class(pricev)
> class(pricev[[1]])
> tail(pricev[[1]])
> # Extract close prices
> pricev <- lapply(pricev, quantmod::Cl)
> # Collapse list into time series the hard way
> prices2 <- cbind(pricev[[1]], pricev[[2]], pricev[[3]], pricev[[4
> class(price2)
> dim(price2)
> # Collapse list into time series using do.call()
> pricev <- do.call(cbind, pricev)
> all.equal(price2, pricev)
> class(pricev)
> dim(pricev)
> # Or extract and cbind in single step
> pricev <- do.call(cbind, lapply(
+   mget(symbolv, envir=rutils::etfenv), quantmod::Cl))
> # Or extract and bind all data, subset by symbolv
> pricev <- lapply(symbolv, function(symboln) {
+     quantmod::Cl(get(symboln, envir=rutils::etfenv))
+ })  # end lapply
> # Or loop over etfenv without anonymous function
> pricev <- do.call(cbind,
+   lapply(as.list(rutils::etfenv)[symbolv], quantmod::Cl))
> # Same, but works only for OHLC series - produces error
> pricev <- do.call(cbind,
+   eapply(rutils::etfenv, quantmod::Cl)[symbolv])
```

Managing Time Series

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

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

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

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

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

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

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

Calculating Percentage Returns from Close Prices

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

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

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

Managing Data Inside Environments

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

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

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

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

Stock Databases And Survivorship Bias

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

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

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

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

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

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

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

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

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

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

Downloading Stock Time Series From *Tiingo*

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

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

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

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

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

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

Coercing Date-time Indices

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

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

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

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

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

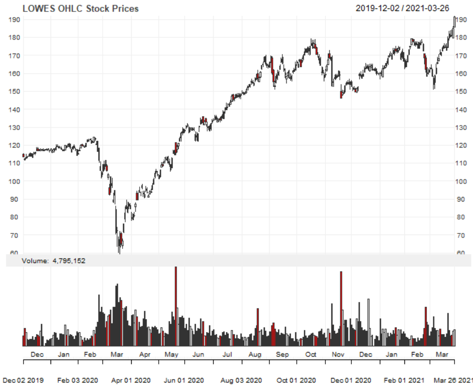
```
> # The date-time index of AAPL is POSIXct
> class(zoo::index(sp500env$AAPL))
> # Coerce the date-time index of AAPL to Date
> zoo::index(sp500env$AAPL) <- as.Date(zoo::index(sp500env$AAPL))
> # Coerce all the date-time indices to Date
> for (symboln in ls(sp500env)) {
+   ohlc <- get(symboln, envir=sp500env)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

Managing Exceptions in Stock Symbols

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

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

```
> # "LOW.Low" is a bad column name
> colnames(sp500env$LOW)
> strsplit(colnames(sp500env$LOW), split="[.]")
> do.call(cbind, strsplit(colnames(sp500env$LOW), split="[.]"))
> do.call(cbind, strsplit(colnames(sp500env$LOW), split="[.]"))[2, ]
> # Extract proper names from column names
> namev <- rutils::get_name(colnames(sp500env$LOW), field=2)
> # Or
> # namev <- do.call(rbind, strsplit(colnames(sp500env$LOW),
> #                                  split="[.]"))[, 2]
> # Rename "LOW" colnames to "LOWES"
> colnames(sp500env$LOW) <- paste("LOWES", namev, sep=".")
> sp500env$LOWES <- sp500env$LOW
> rm(LOW, envir=sp500env)
> # Rename BF-B colnames to "BFB"
> colnames(sp500env$"BF-B") <- paste("BFB", namev, sep=".")
> sp500env$BFB <- sp500env$"BF-B"
> rm("BF-B", envir=sp500env)
> # Rename BRK-B colnames
> sp500env$BRKB <- sp500env$‘BRK-B‘
> rm(‘BRK-B‘, envir=sp500env)
> colnames(sp500env$BRKB) <- gsub("BRK-B", "BRKB", colnames(sp500e
> # Save OHLC prices to .RData file
> save(sp500env, file="/Users/jerzy/Develop/lecture_slides/data/sp500.RData")
> # Download "BRK.B" separately with auto.assign=FALSE
> # BRKB <- quantmod::getSymbols("BRK-B", auto.assign=FALSE, src="tiingo", adjust=TRUE, from="1990-01-01", api.key="j84ac2b9c5bde2d68e3
> # colnames(BRKB) <- paste("BRKB", namev, sep=".")
> # sp500env$BRKB <- BRKB
```



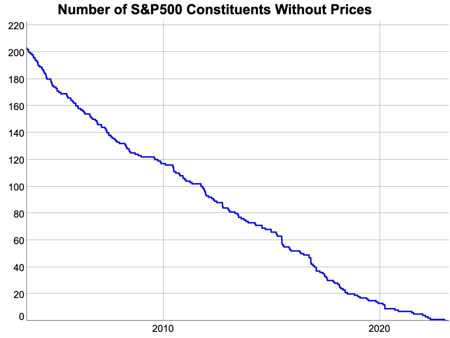
```
> # Plot OHLC candlestick chart for LOWES
> chart_Series(x=sp500env$LOWES["2019-12/"],
+   TA="add_Vo()", name="LOWES OHLC Stock Prices")
> # Plot dygraph
> dygraphs::dygraph(sp500env$LOWES["2019-12/", -5], main="LOWES OHL
+   dyCandlestick()
```

S&P500 Stock Index Constituent Prices

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

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

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

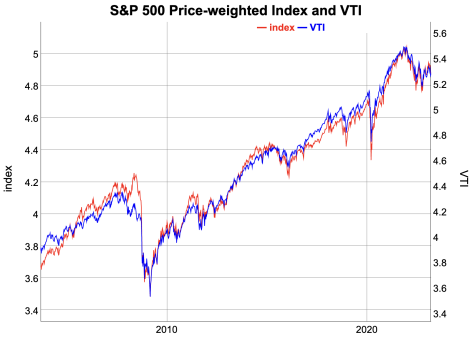


```
> # Calculate number of constituents without prices
> datav <- rowSums(is.na(pricev))
> datav <- xts::xts(datav, order.by=zoo::index(pricev))
> dygraphs::dygraph(datav, main="Number of S&P500 Constituents Witho
+   dyOptions(colors="blue", strokeWidth=2)
```

S&P500 Stock Portfolio Index

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

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



Writing Time Series To Files

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

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

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

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

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

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

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

Reading Time Series from Files

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

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

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

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

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

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

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

Downloading Stock Time Series From *Alpha Vantage*

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

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

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

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

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

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

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

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

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

There are workarounds but they're tedious.

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

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

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

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

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

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

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

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

Yahoo provides daily *OHLC* prices for the *DJIA* index (symbol *ˆDJI*), 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 *ˆDJI* for the *DJIA* stock market index isn't a valid name in `R`.

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

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

Calculating Prices and Returns From *OHLC* Data

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

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

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

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

Downloading Stock Prices From Polygon

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

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

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

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

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

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

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

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

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

Downloading Multiple Stock Prices From Polygon

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

```
> # Select ETF symbols for asset allocation
> symbolv <- c("SPY", "VTI", "QQQ", "VEU", "EEM", "XL
+ "XLE", "XLF", "XLV", "XLI", "XLB", "XLK", "XLU", "V
+ "IWB", "IWD", "IWF", "IEF", "TLT", "VNQ", "DBC", "GI
+ "VXX", "SVXY", "MTUM", "IVE", "VLUE", "QUAL", "VTV"
> # Setup code
> etfenv <- new.env()  # New environment for data
> # Boolean vector of symbols already downloaded
> isdown <- symbolv %in% ls(etfenv)
```

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

Calculating the Stock Alphas, Betas, and Other Performance Statistics

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

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

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

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

draft: Stock Index Weighting Methods

Stock market indices can be capitalization-weighted (*S&P500*), price-weighted (*DJIA*), or equal-weighted.

The cap-weighted and price-weighted indices own a fixed number of shares (excluding stock splits).

Equal-weighted indices own the same dollar amount of each stock, so they must be rebalanced as market prices change.

Cap-weighted index = Sum { (Stock Price * Number of shares) / Index Divisor }

Price-weighted index = Sum {Stock Price / Index Divisor }

Equal-weighted index = Sum { (Stock Price * factor) / Index Divisor }

Cap-weighted indices are over-weight large-cap stocks, while equal-weighted indices are over-weight small-cap stocks.

```
> # Create name corresponding 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 etfenv
> quantmod::getSymbols("SP500", env=etfenv,
+     adjust=TRUE, auto.assign=TRUE, from="1990-01-01")
> chart_Series(x=etfenv$SP500["2016/"],
+        TA="add_Vo()", name="S&P500 index")
```

Scraping *S&P500* Stock Index Constituents From Websites

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

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

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

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

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

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

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

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

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

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

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

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

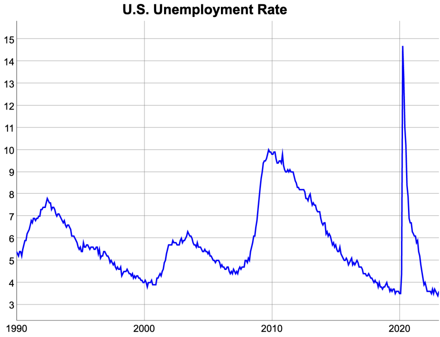
Downloading *FRED* Time Series Data

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

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

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

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



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

The *Quandl* Database

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

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

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

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

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

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

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

Downloading Time Series Data from *Quandl*

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

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

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

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

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

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

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

Stock Index and Instrument Metadata on *Quandl*

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

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

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

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

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

Downloading Multiple Time Series from *Quandl*

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

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

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

Downloading Futures Time Series from *Quandl*

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

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

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

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

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

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

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

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

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

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

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

Downloading *VIX* Futures Files from CBOE

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

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

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

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

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

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

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

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

Downloading *VIX* Futures Data Into an Environment

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

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

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

Reading Data From Excel Files

The package *readxl* reads data from Excel spreadsheet files into R.

The function read_excel() reads a single sheet (tab) from an Excel file.

The function read_xlsx() reads a single sheet (tab) from an Excel file in .xlsx format.

The functions from package *readxl* return a type of *data frame* called a *tibble* object.

The *tibble* classes tbl and tbl_df are derived from the *data frame* class data.frame.

tibble objects are also used by the package *dplyr*.

DataCamp offers a Tutorial on Importing Excel Files into R.

```
> # Install and load package readxl
> install.packages("readxl")
> library(readxl)
> dirn <- "/Users/jerzy/Develop/lecture_slides/data"
> filev <- file.path(dirn, "multi_tabs.xlsx")
> # Read a time series from first sheet of xlsx file
> tibblev <- readxl::read_xlsx(filev)
> class(tibblev)
> # Coerce POSIXct dates into Date class
> class(tibblev$Dates)
> tibblev$Dates <- as.Date(tibblev$Dates)
> # Some columns are character strings
> sapply(tibblev, class)
> sapply(tibblev, is.character)
> # Coerce columns with strings to numeric
> listv <- lapply(tibblev, function(x) {
+   if (is.character(x))
+     as.numeric(x)
+   else
+     x
+ })  # end lapply
> # Coerce list into xts time series
> xtsv <- xts::xts(do.call(cbind, listv)[, -1], listv[[1]])
> class(xtsv); dim(xtsv)
> # Replace NA values with the most recent non-NA values
> sum(is.na(xtsv))
> xtsv <- zoo::na.locf(xtsv, na.rm=FALSE)
> xtsv <- zoo::na.locf(xtsv, fromLast=TRUE)
```

Reading Multiple Sheets From Excel Files

The function readxl::excel_sheets() returns a vector of character strings with the names of all the sheets in an Excel spreadsheet.

The package *readxl* reads data from Excel spreadsheet files into R.

The function read_excel() reads a single sheet (tab) from an Excel file.

The function read_xlsx() reads a single sheet (tab) from an Excel file in .xlsx format.

The functions from package *readxl* return a type of *data frame* called a *tibble* object.

The *tibble* classes tbl and tbl_df are derived from the *data frame* class data.frame.

tibble objects are also used by the package *dplyr*.

```
> # Read names of all the sheets in an Excel spreadsheet
> namev <- readxl::excel_sheets(filev)
> # Read all the sheets from an Excel spreadsheet
> sheetv <- lapply(namev, read_xlsx, path=filev)
> names(sheetv) <- namev
> # sheetv is a list of tibbles
> sapply(sheetv, class)
> # Create function to coerce tibble to xts
> to_xts <- function(tibblev) {
+   tibblev$Dates <- as.Date(tibblev$Dates)
+   # Coerce columns with strings to numeric
+   listv <- lapply(tibblev, function(x) {
+     if (is.character(x))
+       as.numeric(x)
+     else
+       x
+   })  # end lapply
+   # Coerce list into xts series
+   xts::xts(do.call(cbind, listv)[, -1], listv$Dates)
+ }  # end to_xts
> # Coerce list of tibbles to list of xts
> class(sheetv)
> sheetv <- lapply(sheetv, to_xts)
> sapply(sheetv, class)
> # Replace NA values with the most recent non-NA values
> sapply(sheetv, function(xtsv) sum(is.na(xtsv)))
> sheetv <- lapply(sheetv, zoo::na.locf, na.rm=FALSE)
> sheetv <- lapply(sheetv, zoo::na.locf, fromLast=TRUE)
```

Performing Calculations in Excel Using R

Excel can run R using either VBA scripts, or through a *COM* interface (available on *Windows* only).

R can perform calculations and export its output to Excel files, or it can modify Excel files (requires packages using Java or Perl code).

Calculations in R and Excel can be combined in several different ways:

- Data from Excel can be exchanged with R via .csv files (simplest and best method),
- Excel can execute R commands using VBA scripts, and then import the R output from .csv files,
- An Excel add-in can execute R commands as Excel functions (relies on *COM* protocol, so works only for *Windows*): add-ins *BERT*, *RExcel*,
- R can modify Excel files and run Excel functions (requires packages using Java or Perl code): packages *xlsx*, *XLConnect*, *excel.link*,

```
> ### Perform calculations in R,
> ### And export to CSV files
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> # Subset data frame
> readf <- readf[readf[, "type"]=="daisy", ]
> all.equal(readf, dframe)
> # Write data frame to CSV file, with row names
> write.csv(readf, file="daisies.csv")
```

Running R Code from Excel

There are several ways of performing calculations in R and exporting the outputs to Excel:

- Export data from Excel via .csv files to R, perform the calculations in R, and import the outputs back to Excel via .csv files (simplest and best method),
- Run R from Excel using VBA scripts, and exchange data via .csv files,
- Run R from Excel using an Excel add-in, and execute R commands as Excel functions (relies on the *COM* protocol, so works only for *Windows*),

```
> ### Perform calculations in R,
> ### And export to CSV files
> setwd("/Users/jerzy/Develop/lecture_slides/data")
> # Read data frame, with row names from first column
> readf <- read.csv(file="florist.csv", row.names=1)
> # Subset data frame
> readf <- readf[readf[, "type"]=="daisy", ]
> all.equal(readf, dframe)
> # Write data frame to CSV file, with row names
> write.csv(readf, file="daisies.csv")
```

Running R Code Using VBA Scripts

An R session can be launched from Excel using a VBA script (macro).

The VBA function shell() executes a program by running an executable *exe* file (with extension *exe*).

A VBA script can also run an R *batch* process.

The R *batch* process can write to .csv files, which can then be imported into Excel.

```
' VBA macro to run R process
Sub run_r()
 Call shell("R", vbNormalFocus)
End Sub
```

```
' VBA macro to run interactive R process
Sub run_rinteractive()
 Dim script_dir As String: script_dir = "C:\Develop\R\script
 Dim script_file As String: script_file = "plot_interactive.
 Dim log_file As String: log_file = "C:\Develop\R\scripts\lo
 Call shell("R --vanilla < " & script_dir & script_file & ">
End Sub
```

```
' VBA macro to run batch R process
Sub run_rbatch()
 Dim script_dir As String: script_dir = "C:\Develop\R\script
 Dim script_file As String: script_file = "plot_to_file.R"
 Dim log_file As String: log_file = "C:\Develop\R\scripts\lo
 Call shell("R --vanilla < " & script_dir & script_file & ">
End Sub
```

BERT Excel Add-in for Running R Code

BERT is an Excel add-in which allows executing R commands as Excel functions:
http://bert-toolkit.com/
http://bert-toolkit.com/bert-quick-start
https://github.com/sdllc/Basic-Excel-R-Toolkit/wiki
https://github.com/sdllc/Basic-Excel-R-Toolkit

BERT launches its own R process from Excel.

BERT can create its own menu in the Excel add-ins tab:
After installing *BERT*, click on upper-left *Office Button*, click Excel options, on the bottom of the window choose (Manage: *COM* Add-ins) Go, add the *COM* add-in BERTRibbon2x86.dll.

BERT relies on the *COM* protocol, so it works only for *Windows*.

```
' calculate sum of Excel cells using R
R.Add(B1:D1)

' remove NAs over Excel cell range using R function
R.na_omit(F2:H4)

' calculate eigenValues of Excel matrix using R function
R.EigenValues(A1:H8)
```

Package *googlesheets* for Interacting with *Google Sheets*

The package *googlesheets* allows interacting with *Google Sheets* using R commands.

If you already have a *Google* account, then your personal *Google Sheets* can be found at:
https://docs.google.com/spreadsheets/

The function gs_ls() listv the files in *Google Sheets*.

The function gs_title() registers a *Google* sheet, and returns a googlesheet object.

A googlesheet object contains information (metadata) about a *Google* sheet, such as its name and key, but not the sheet data itself.

The function gs_browse() opens a *Google* sheet in an internet browser.

You can find online a document about using *googlesheets*.

You can find online a document about managing authentication tokens.

```
> # Install latest version of googlesheets
> devtools::install_github("jennybc/googlesheets")
> # Load package googlesheets
> library(googlesheets)
> library(dplyr)
> # Authenticate authorize R to view and manage your files
> gs_auth(new_user=TRUE)
> # List the files in Google Sheets
> googlesheets::gs_ls()
> # Register a sheet
> googsheet <- gs_title("my_data")
> # view sheet summary
> googsheet
> # List tab names in sheet
> tabv <- gs_ws_ls(googsheet)
> # Set curl options
> library(httr)
> httr::set_config(config(ssl_verifypeer=0L))
> # Read data from sheet
> gs_read(googsheet)
> # Read data from single tab of sheet
> gs_read(googsheet, ws=tabv[1])
> gs_read_csv(googsheet, ws=tabv[1])
> # Or using dplyr pipes
> googsheet %>% gs_read(ws=tabv[1])
> # Download data from sheet into file
> gs_download(googsheet, ws=tabv[1],
+       to="/Users/jerzy/Develop/lecture_slides/data/googsheet.csv"
> # Open sheet in internet browser
> gs_browse(googsheet)
```

draft: Downloading Data from *Google Sheets*

The package *googlesheets* allows interacting with *Google Sheets* from R.

If you already have a *Google* account, then your personal *Google Sheets* can be found at:
https://docs.google.com/spreadsheets/

The function gs_ls() listv the files in *Google Sheets*.

The function gs_title() registers a *Google* sheet, and returns a googlesheet object.

A googlesheet object contains information (metadata) about a *Google* sheet, such as its name and key, but not the sheet data itself.

The function gs_read() downloads data from a *Google* sheet and returns a data frame.

The function gs_download() downloads data from a *Google* sheet into a file.

The function gs_browse() opens a *Google* sheet in an internet browser.

```
> # Install latest version of googlesheets
> devtools::install_github("jennybc/googlesheets")
> # Load package googlesheets
> library(googlesheets)
> library(dplyr)
> # Authenticate authorize R to view and manage your files
> gs_auth(new_user=TRUE)
> # List the files in Google Sheets
> googlesheets::gs_ls()
> # Register a sheet
> googsheet <- gs_title("my_data")
> # view sheet summary
> googsheet
> # List tab names in sheet
> tabv <- gs_ws_ls(googsheet)
> # Set curl options
> library(httr)
> httr::set_config(config(ssl_verifypeer=0L))
> # Read data from sheet
> gs_read(googsheet)
> # Read data from single tab of sheet
> gs_read(googsheet, ws=tabv[1])
> gs_read_csv(googsheet, ws=tabv[1])
> # Or using dplyr pipes
> googsheet %>% gs_read(ws=tabv[1])
> # Download data from sheet into file
> gs_download(googsheet, ws=tabv[1],
+         to="/Users/jerzy/Develop/lecture_slides/data/googsheet.csv"
> # Open sheet in internet browser
> gs_browse(googsheet)
```