FRE6871 R in Finance
Lecture#1, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 5, 2022

Welcome Students!

My name is Jerzy Pawlowski jp3900@nyu.edu

I'm an adjunct professor at NYU Tandon because I love teaching and I want to share my professional knowledge with young, enthusiastic students.
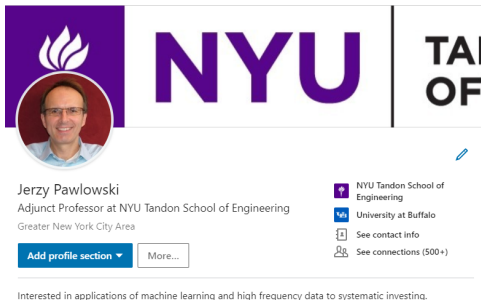
I'm interested in applications of *machine learning* to *systematic investing*.

I'm an advocate of *open-source software*, and I share it on GitHub:
My GitHub account

In my finance career, I have worked as a hedge fund *portfolio manager*, *CLO structurer* (banker), and *quant analyst*.
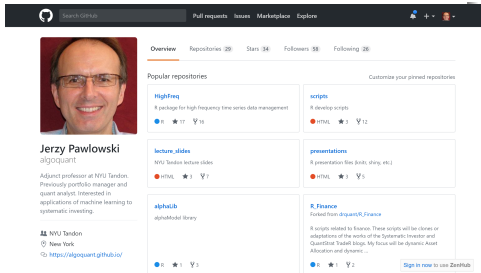My LinkedIn profile

FRE6871 Course Description and Objectives

Course Description

The course will study the applications of the R statistical language to financial data analysis and modeling. The applications will include *classification* for credit scoring, *Monte Carlo simulation* for option pricing and credit portfolio modeling, and *Principal Component Analysis* (*PCA*) for interest rate yield curve modeling. The course will apply statistical techniques, such as *hypothesis testing*. *linear regression*, *logistic regression*, and *bootstrap simulation*.

FRE6871 Course Description and Objectives

Course Description

The course will study the applications of the R statistical language to financial data analysis and modeling. The applications will include *classification* for credit scoring, *Monte Carlo simulation* for option pricing and credit portfolio modeling, and *Principal Component Analysis* (*PCA*) for interest rate yield curve modeling. The course will apply statistical techniques, such as *hypothesis testing*, *linear regression*, *logistic regression*, and *bootstrap simulation*.

Course Objectives

Students will learn through R coding exercises how to:
 - Manipulate data structures (vectors, data frames, dates, and time series).
 - Download data from external sources, and to scrub and format it.
 - Create interactive plots and visualizations.
 - Build financial models.
 - Perform exception and error handling, and debugging.

FRE6871 Course Description and Objectives

Course Description

The course will study the applications of the `R` statistical language to financial data analysis and modeling. The applications will include *classification* for credit scoring, *Monte Carlo simulation* for option pricing and credit portfolio modeling, and *Principal Component Analysis* (*PCA*) for interest rate yield curve modeling. The course will apply statistical techniques, such as *hypothesis testing. linear regression*, *logistic regression*, and *bootstrap simulation*.

Course Objectives

Students will learn through `R` coding exercises how to:
- Manipulate data structures (vectors, data frames, dates, and time series).
- Download data from external sources, and to scrub and format it.
- Create interactive plots and visualizations.
- Build financial models.
- Perform exception and error handling, and debugging.

Course Prerequisites

The `R` language is considered to be challenging, so this course requires some programming experience with other languages such as `C++` or `Python`. Students should also have knowledge of basic statistics (random variables, estimators, hypothesis testing, regression, etc.) The course *FRE7241 Algorithmic Portfolio Management* is designed as a followup course to *FRE6871*.

Homeworks and Tests

Homeworks and Tests

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

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

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

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

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

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

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

Homeworks and Tests

Homeworks and Tests

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

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

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

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

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

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

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

Graduate Assistant

The graduate assistant (GA) will be Shardha Koul (sk9225@nyu.edu).
The GA will answer questions during office hours, or via *Brightspace* forums, not via emails. Please send emails regarding lecture matters from *Brightspace* (not personal emails).

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

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

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

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

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

Tips for Solving Homeworks and Tests

Tips for Solving Homeworks and Tests

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

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

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

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

Please Submit *Minimal Working Examples* With Your Questions

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

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

You can read more about producing *minimal working examples* here: http://stackoverflow.com/help/mcve
http://www.jaredknowles.com/journal/2013/5/27/writing-a-minimal-working-example-mwe-in-r

Course Grading Policies

Numerical Scores

Homeworks and tests will be graded and assigned numerical scores. Each part of homeworks and tests will be graded separately and assigned a numerical score.

Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.

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

Course Grading Policies

Numerical Scores

Homeworks and tests will be graded and assigned numerical scores. Each part of homeworks and tests will be graded separately and assigned a numerical score.

Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.

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

Letter Grades

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

Course Grading Policies

Numerical Scores

Homeworks and tests will be graded and assigned numerical scores. Each part of homeworks and tests will be graded separately and assigned a numerical score.

Maximum scores will be given only for complete code, that produces the correct output when it's pasted into an R session, without any modifications. As long as the R code uses the required functions and produces the correct output, it will be given full credit.

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

Letter Grades

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

Plagiarism

Plagiarism (copying from other students) and cheating will be punished.

But copying code from lecture slides, books, or any online sources is allowed and encouraged.

Students must provide references to any external sources from which they copy code (such as links or titles and page numbers).

FRE6871 Course Materials

Lecture Slides

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

FRE6871 Course Materials

Lecture Slides

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

FRE6871 Recommended Textbooks

- *Statistics and Data Analysis for Financial Engineering* by David Ruppert, introduces regression, cointegration, multivariate time series analysis, *ARIMA*, *GARCH*, *CAPM*, and factor models, with examples in R.

- *Quantitative Risk Management* by Alexander J. McNeil, Rudiger Frey, and Paul Embrechts: review of Value at Risk, factor models, ARMA and GARCH, extreme value theory, and credit risk models.

- *Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, introduces machine learning techniques using R, but without deep learning.

- *Advanced R* by Hadley Wickham, is the best book for learning the advanced features of R.

- *The Art of R Programming* by Norman Matloff, contains a good introduction to R and to some statistical models.

Many textbooks can be downloaded in electronic format from the *NYU Library*.

FRE6871 Supplementary Textbooks

Supplementary Textbooks

- The books *R in Action* by Robert Kabacoff and *R for Everyone* by Jared Lander, are good introductions to R and to statistical models.

- *Applied Econometrics with R* by Christian Kleiber and Achim Zeileis, introduces advanced statistical models and econometrics.

- *Numerical Recipes in C++* by William Press, Saul Teukolsky, William Vetterling, and Brian Flannery, is a great reference for linear algebra and numerical methods, implemented in working C++ code.

FRE6871 Supplementary Materials

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

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

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

Internal R Help and Documentation

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

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

```
> # Display documentation on function "getwd"
> help(getwd)
> # Equivalent to "help(getwd)"
> ?getwd
```

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

R documentation is also available in `RGui` under the help tab.

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

```
> # Open the hypertext documentation
> help.start()
```


Introduction to R by Venables and R Core Team.

R Online Help and Documentation

R Cheat Sheets

Cheat Sheets are a fast way to find what you want
https://www.rstudio.com/resources/cheatsheets/

R Programming Wikibook

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

R FAQ

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

R-seek Online Search Tool

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

R-help Mailing List

R-help is a very comprehensive Q&A mailing list
https://stat.ethz.ch/mailman/listinfo/r-help
R-help has archives of past Q&A - search it before you ask
https://stat.ethz.ch/pipermail/r-help/

R Code Style Guidelines

Please follow the *RStudio Style Guide* to make your R code more readable.

Please also follow these R code style rules:

- Use the left arrow "<-" for assignment, not the equals sign "=" (to insert "<-" into code, use the *Alt-underscore* shortcut in Windows),
- Use *nouns* for variable names and *verbs* for function names,
- Use a combination of lowercase letters, numbers, and underscores "_" for names of variables and functions,
- Add underscores "_" to names to avoid conflicts with the names of existing R functions and variables,
- Do not use dots "." in names, except in the names of function *methods* (even though R uses them for variables as well),
- Use underscores "_" in file names, instead of spaces,
- Always put a space after a comma, never before it: "x, y" not "x , y",
- Do not put spaces inside or outside parentheses: "if (x > 0)" not "if (x > 0)",
- Surround infix operators (==, +, -, <-, etc.) with spaces: "x > 0" not "x>0" (even though I don't always follow that rule, to save whitespace),
- Add a comment after the closing curly bracket: "} # end my_fun",

You can reformat R code chunks using the *styler* macros in the *RStudio Addins* drop-down menu.
You can also reformat whole files with R code by using the *styler* package.

Stack Exchange

Stack Overflow

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

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

Stack Exchange

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

http://stackexchange.com/
http://stackexchange.com/sites#technology
http://quant.stackexchange.com/
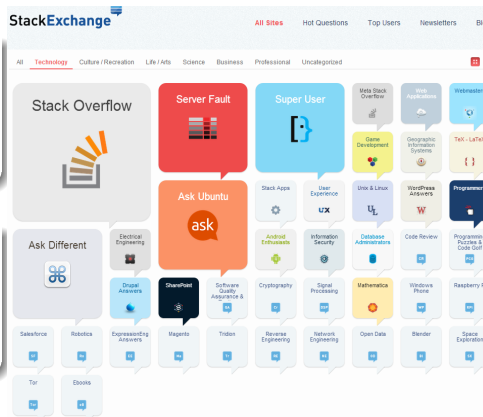
RStudio Support

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

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

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

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

R Online Books and References

Hadley Wickham book *Advanced R*

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

Cookbook for R by Winston Chang from *RStudio*

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

Efficient R programming by Colin Gillespie and Robin Lovelace

Good tips for fast R programming: https://csgillespie.github.io/efficientR/programming.html

Endmemo web book

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

Quick-R by Robert Kabacoff

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

R for Beginners by Emmanuel Paradis

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

R Online Interactive Courses

Datacamp Interactive Courses

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

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

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

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

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

Try R

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

R Blogs and Experts

R-Bloggers

R-Bloggers is an aggregator of blogs dedicated to R
http://www.r-bloggers.com/
Tal Galili is the author of R-Bloggers and has his own excellent blog
http://www.r-statistics.com/

Dirk Eddelbuettel

Dirk is a *Top Answerer* for R questions on Stackoverflow, the author of the Rcpp package, and the CRAN Finance View
http://dirk.eddelbuettel.com/
http://dirk.eddelbuettel.com/code/
http://dirk.eddelbuettel.com/blog/
http://www.rinfinance.com/

Romain Frangois

Romain is an R Enthusiast and Rcpp Hero
http://romainfrancois.blog.free.fr/
http://romainfrancois.blog.free.fr/index.php?tag/graphgallery
http://blog.r-enthusiasts.com/

More R Blogs and Experts

Revolution Analytics Blog

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

RStudio Blog

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

GitHub for Hosting Software Projects Online

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

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

Most R projects are now hosted on *GitHub*.

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

https://github.com/tensorflow/tensorflow

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

https://github.com/algoquant/lecture_slides
https://github.com/algoquant

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

What is R?

- An open-source software environment for statistical computing and graphics.
- An interpreted language, that allows interactive code development.
- A functional language where every operator is an R function.
- A very expressive language that can perform complex operations with very few lines of code.
- A language with metaprogramming facilities that allow programming on the language.
- A language written in C/C++, which can easily call other C/C++ programs.
- Can be easily extended with *packages* (function libraries), providing the latest developments like *Machine Learning*.
- Supports object-oriented programming with *classes* and *methods*.
- Vectorized functions written in C/C++, allow very fast execution of loops over vector elements.

Project

Wikipedia

Why is R More Difficult Than Other Languages?

R is more difficult than other languages because:

- R is a *functional* language, which makes its syntax unfamiliar to users of procedural languages like C/C++.

- The huge number of user-created *packages* makes it difficult to tell which are the best for particular applications.

- R can produce very cryptic *warning* and *error* messages, because it's a programming environment, so it performs many operations quietly, but those can sometimes fail.

- Fixing errors usually requires analyzing the complex structure of the R programming environment.

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

What are the Best Ways to Use R?

If used properly, R can be fast and interactive:

- Use R as an interface to libraries written in C++, Java, and JavaScript.
- Avoid using too many R function calls (every command in R is a function).
- Avoid using apply() and for() loops for large datasets.
- Use R functions which are *compiled* C++ code, instead of using interpreted R code.
- Use package *data.table* for high performance data management.
- Use package *shiny* for interactive charts of live models running in R.
- Use package *dygraphs* for interactive time series plots.
- Use package *knitr* for *RMarkdown* documents.
- Pre-allocate memory for new objects.
- Write C++ functions in *Rcpp* and *RcppArmadillo*.


```
> # Calculate cumulative sum of a vector
> vectorv <- runif(1e5)
> # Use compiled function
> cumsumv <- cumsum(vectorv)
> # Use for loop
> cumsumv2 <- vectorv
> for (i in 2:NROW(vectorv))
+   cumsumv2[i] <- (cumsumv2[i] + cumsumv2[i-1])
> # Compare the two methods
> all.equal(cumsumv, cumsumv2)
> # Microbenchmark the two methods
> library(microbenchmark)
> summary(microbenchmark(
+   cumsum=cumsum(vectorv),
+   loop=for (i in 2:NROW(vectorv))
+     vectorv[i] <- (vectorv[i] + vectorv[i-1]),
+   times=10))[, c(1, 4, 5)]
```

The R License

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

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

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

http://creativecommons.org

Installing R and *RStudio*

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

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

Students will be required to install and to become proficient with the R Interpreter,
Students can download the R Interpreter from CRAN (Comprehensive R Archive Network):
http://cran.r-project.org/

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

Students will be required to install and to become proficient with the *RStudio* Integrated Development Environment (*IDE*),
http://www.rstudio.com/products/rstudio/

Using *RStudio*

A First R Session

Variables are created by an assignment operation, and they don't have to be declared.

The standard assignment operator in R is the arrow symbol "<-".

R interretsp text in quotes ("") as character strings.

Text that is not in quotes ("") is interpreted as a *symbol* or *expression*.

Typing a *symbol* or *expression* evaluates it.

R uses the hash "#" sign to mark text as comments.

All text after the hash "#" sign is treated as a comment, and is not executed as code.

```
> # "<-" and "=" are valid assignment operators
> myvar <- 3
>
> # typing a symbol or expression evaluates it
> myvar
[1] 3
>
> # text in quotes is interpreted as a string
> myvar <- "Hello World!"
>
> # typing a symbol or expression evaluates it
> myvar
[1] "Hello World!"
>
> myvar  # text after hash is treated as comment
[1] "Hello World!"
```

Exploring an R Session

The function `getwd()` returns a vector of length 1, with the first element containing a string with the name of the current working directory (`cwd`).

```
> getwd()  # get cwd
> setwd("/Users/jerzy/Develop/R")  # Set cwd
> getwd()  # get cwd
```

The function `setwd()` accepts a character string as input (the name of the directory), and sets the working directory to that string.

R is a functional language, and R commands are functions, so they must be followed by parentheses "()".

Get system date and time

Just the date

```
> Sys.time()  # get date and time
[1] "2022-09-05 17:13:52 EDT"
>
> Sys.Date()  # get date only
[1] "2022-09-05"
```

The R Workspace

The workspace is the current R working environment, which includes all user-defined objects and the command history.

The function `ls()` returns names of objects in the R workspace.

The function `rm()` removes objects from the R workspace.

The workspace can be saved into and loaded back from an `.RData` file (compressed binary file format).

The function `save.image()` saves the whole workspace.

The function `save()` saves just the selected objects.

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

```
> var1 <- 3  # Define new object
> ls()  # List all objects in workspace
> # List objects starting with "v"
> ls(pattern=glob2rx("v*"))
> # Remove all objects starting with "v"
> rm(list=ls(pattern=glob2rx("v*")))
> save.image()  # Save workspace to file .RData in cwd
> rm(var1)  # Remove object
> ls()  # List objects
> load(".RData")
> ls()  # List objects
> var2 <- 5  # Define another object
> save(var1, var2,  # Save selected objects
+      file="/Users/jerzy/Develop/lecture_slides/data/my_data.RData"
> rm(list=ls())  # Remove all objects
> ls()  # List objects
> load_ed <- load(file="/Users/jerzy/Develop/lecture_slides/data/my_
> load_ed
> ls()  # List objects
```

The R Workspace (cont.)

When you quit R you'll be prompted "Save workspace image?"

If you answer *YES* then the workspace will be saved into the `.RData` file in the `cwd`,

When you start R again, the workspace will be automatically loaded from the existing `.RData` file,

```
> q()  # quit R session
```

The function `history()` displays recent commands,

You can also save and load the command history from a file.

```
> history(5)  # Display last 5 commands
> savehistory(file="myfile")  # Default is ".Rhistory"
> loadhistory(file="myfile")  # Default is ".Rhistory"
```

R Session Info

The function `sessionInfo()` returns information about the current R session,

- R version,
- OS platform,
- locale settings,
- list of packages that are loaded and attached to the search path,
- list of packages that are loaded, but *not* attached to the search path,

```
> sessionInfo()  # get R version and other session info
R version 4.2.1 (2022-06-23)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS Monterey 12.5.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources
LAPACK: /Library/Frameworks/R.framework/Versions/4.2-arm64/Resources

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

attached base packages:
[1] graphics  grDevices utils     datasets  stats     methods   base

other attached packages:
[1] knitr_1.37      HighFreq_0.1    rutils_0.2      dygraphs_1.1.
[5] quantmod_0.4.20 TTR_0.24.3      xts_0.12.1      zoo_1.8-10

loaded via a namespace (and not attached):
 [1] Rcpp_1.0.9       lattice_0.20-45  digest_0.6.29    grid_4.2.
 [5] magrittr_2.0.3   evaluate_0.14    highr_0.9        stringi_1.
 [9] rlang_1.0.5      cli_3.3.0        curl_4.3.2       rstudioap
[13] tools_4.2.1      stringr_1.4.0    htmlwidgets_1.5.4 xfun_0.29
[17] compiler_4.2.1   fastmap_1.1.0    htmltools_0.5.3
```

Environment Variables

R uses environment variables to store information about its environment, such as paths to directories containing files used by R (startup, history, OS).

For example the environment variables:

- R_USER and HOME store the R user Home directory,
- R_HOME stores the root directory of the R installation,

The functions Sys.getenv() and Sys.setenv() display and set the values environment variables.

Sys.getenv("env_var") displays the environment variable "env_var".

Sys.setenv("env_var=value") sets the environment variable "env_var" equal to "value".

```
> Sys.getenv()[5:7]  # List some environment variables
>
> Sys.getenv("HOME")  # get R user HOME directory
>
> Sys.setenv(Home="/Users/jerzy/Develop/data")  # Set HOME directory
>
> Sys.getenv("HOME")  # get user HOME directory
>
> Sys.getenv("R_HOME")  # get R_HOME directory
>
> R.home()  # get R_HOME directory
>
> R.home("etc")  # get "etc" sub-directory of R_HOME
```

Global *Options* Settings

R uses a list of global *options* which affect how R computes and displays results.

The function options() either sets or displays the values of global *options*.

options("globop") displays the current value of option "globop".

getOption("globop") displays the current value of option "globop".

options(globop=value) sets the option "globop" equal to "value".

```
> # ?options  # Long list of global options
> # Interpret strings as characters, not factors
> getOption("stringsAsFactors")  # Display option
> options("stringsAsFactors")  # Display option
> options(stringsAsFactors=FALSE)  # Set option
> # number of digits printed for numeric values
> options(digits=3)
> # control exponential scientific notation of print method
> # positive "scipen" values bias towards fixed notation
> # negative "scipen" values bias towards scientific notation
> options(scipen=100)
> # maximum number of items printed to console
> options(max.print=30)
> # Warning levels options
> # negative - warnings are ignored
> options(warn=-1)
> # zero - warnings are stored and printed after top-confl function
> options(warn=0)
> # One - warnings are printed as they occur
> options(warn=1)
> # two or larger - warnings are turned into errors
> options(warn=2)
> # Save all options in variable
> op_tions <- options()
> # Restore all options from variable
> options(op_tions)
```

Constructing File Paths

Names of *file paths* can be constructed using the function `paste()`.

The function `file.path()` is similar to `paste()`, but it also automatically uses the correct file separator for the computer platform.

The function `normalizePath()` performs tilde-expansions and displays file paths in user-readable format.

```
> # R startup (site) directory
> paste(R.home(), "etc", sep="/")
[1] "/Library/Frameworks/R.framework/Resources/etc"
>
> file.path(R.home(), "etc")  # better way
[1] "/Library/Frameworks/R.framework/Resources/etc"
>
> # perform tilde-expansions and convert to readable format
> normalizePath(file.path(R.home(), "etc"), winslash="/")
[1] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/et
>
> normalizePath(R.home("etc"), winslash="/")
[1] "/Library/Frameworks/R.framework/Versions/4.2-arm64/Resources/et
```

R System Directories under *Windows*

R uses several different directories to search, read, and store files:

- *Windows* user `personal` directory: `"~"` (`"%USERPROFILE%/Documents"`),
- R user `HOME` directory (`R_USER` and `Home`),
- `cwd` current working directory - the default directory for storing and retrieving user files (such as `.Rhistory`, `.RData`, etc.),
- `R_HOME` root directory of the R installation,
- R startup (site) directory: `R_HOME/etc/`,

By default, the R user `HOME` directory is the *Windows* user `personal` directory.

The `cwd` is set to the directory from which R is invoked, or the R user `HOME` directory.

```
> normalizePath("~", winslash="/")  # Windows user HOME directory
>
> Sys.getenv("HOME")  # R user HOME directory
>
> setwd("/Users/jerzy/Develop/R")
> getwd()  # current working directory
>
> # R startup (site) directory
> normalizePath(file.path(R.home(), "etc"), winslash="/")
>
> # R executable directory
> normalizePath(file.path(R.home(), "bin/x64"), winslash="/")
>
> # R documentation directory
> normalizePath(file.path(R.home(), "doc/manual"), winslash="/")
```

File and Directory Listing Functions

The functions `list.files()` and `dir()` return a vector of names of files in a given directory.

The function `list.dirs()` listv the directories in a given directory.

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

```
> sample(dir(), 5)  # get 5 file names - dir() listv all files
> sample(dir(pattern="csv"), 5)  # List files containing "csv"
> sample(list.files(R.home()), 5)  # All files in R_HOME directory
> sample(list.files(R.home("etc")), 5)  # All files in "etc" sub-dir
> sample(list.dirs(), 5)  # Directories in cwd
> list.dirs(R.home("etc"))  # Directories in "etc" sub-directory
> sample(Sys.glob("*.csv"), 5)
> Sys.glob(R.home("etc"))
```

Invoking an R Session in *Windows*

An R session can run in several different ways:

- In an R terminal (by invoking `R.exe` or `Rterm.exe`),
- In an R RGui (by invoking `RGui.exe`),
- In an *RStudio* session (or some other IDE),

The initial value of the `cwd` depends on how the R session is invoked.
If R is invoked:

- from the *Windows* menu, then `cwd` is set to the R user `HOME` directory,
- by clicking on a file (`*.R`, `.RData`, etc.), then `cwd` is set to the file's directory,
- by typing `R.exe` or `Rterm.exe` in the command shell (after setting the `PATH`), then `cwd` is set to the directory where the command was typed,

```
> getwd() # get cwd
[1] "/Users/jerzy/Develop/lecture_slides"
```

R Session Startup

At startup R sources (reads) several types of files, in the following order:

- Renviron files defining environment variables,
- Rprofile files containing code executed at R startup,
- RData files containing data to be loaded at R startup,

R sources files from several directories, in the following order:

- R startup directory: Renviron.site and Rprofile.site files,
- cwd directory: .Renviron, .Rprofile, and .RData files,
- HOME user directory (only if no files found in cwd),

The above startup process can be customized by setting environment variables.

```
> # help(Startup)  # Description of R session startup mechanism
>
> # files in R startup directory directory
> dir(normalizePath(file.path(R.home(), "etc"), winslash="/"))
>
> # *.R* files in cwd directory
> getwd()
> dir(getwd(), all.files=TRUE, pattern="\\.R")
> dir(getwd(), all.files=TRUE, pattern=glob2rx("*.R*"))
```

Data Objects in R

All data objects in R are *vectors*, or consist of *vectors*.

Single numbers and character strings are vectors of length "1".

Atomic vectors are *homogeneous* objects whose elements are all of the same *mode* (type).

Lists and *data frames* are *recursive* (heterogeneous) objects, whose elements can be vectors of different *mode*.

The functions is.atomic() and is.recursive() return logical values depending on whether their arguments are *atomic* or *recursive*.

R Data Objects

	Atomic	*Recursive*
1-dim	Vectors	Lists
2-dim	Matrices	Data frames
n-dim	Arrays	NA

```
> # Single numbers are vectors of length 1
> 1
[1] 1
> # Character strings are vectors of length 1
> "a"
[1] "a"
> # Strings without quotes are variable names
> a  # Variable "a" doesn't exist

Error in eval(expr, envir, enclos):  object 'a' not found

> # List elements can have different mode
> list(aa=c("a", "b"), bb=1:5)
$aa
[1] "a" "b"

$bb
[1] 1 2 3 4 5
> data.frame(aa=c("a", "b"), bb=1:2)
  aa bb
1  a  1
2  b  2
> is.atomic(data.frame(aa=c("a", "b"), bb=1:2))
[1] FALSE
> is.recursive(data.frame(aa=c("a", "b"), bb=1:2))
[1] TRUE
```

Type, Mode, and Class of Objects

The *type*, *mode*, and *class* are `character` strings representing various object properties.

The *type* of an atomic object represents how it's stored in memory (`logical`, `character`, `integer`, `double`, etc.)

The *mode* of an atomic object is the kind of data it represents (`logical`, `character`, `numeric`, etc.)

The *mode* of an object often coincides with its *type* (except for `integer` and `double` types).

Recursive objects (`listv`, `data` frames) have both *type* and *mode* equal to the recursive type (`list`).

The *class* of an object is given by either an explicit *class* attribute, or is derived from the object *mode* and its `dim` attribute (implicit *class*).

The function `class()` returns the explicit or implicit *class* of an object.

The object *class* is used for method dispatching in the S3 object-oriented programming system in R.

```
> myvar <- "hello"
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "character" "character" "character"
>
> myvar <- 1:5
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "integer"
>
> myvar <- runif(5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "numeric"
>
> myvar <- matrix(1:10, 2, 5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "matrix"  "array"
>
> myvar <- matrix(runif(10), 2, 5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "matrix"  "array"
>
> myvar <- list(aa=c("a", "b"), bb=1:5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "list" "list" "list"
>
> myvar <- data.frame(aa=c("a", "b"), bb=1:2)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "list"     "list"     "data.frame"
```

R Object Attributes

R objects can have different attributes, such as: namesv, dimnames, dimensions, *class*, etc.

The attributes of an object is a named list of symbol=value pairs.

The function attributes() returns the attributes of an object.

The attributes of an R object can be modified using the "attributes() <-" assignment.

The function structure() adds attributes (specified as symbol=value pairs) to an object, and returns it.

A vector that is assigned an attribute other than namesv is not treated as a vector.

The function is.vector() returns TRUE if its argument is a vector, and returns FALSE otherwise.

```
> # A simple vector has no attributes
> attributes(5:10)
NULL
> myvar <- c(pi=pi, euler=exp(1), gamma=-digamma(1))
> # Named vector has "namesv" attribute
> attributes(myvar)
$names
[1] "pi"    "euler" "gamma"
> myvar <- 1:10
> is.vector(myvar)  # Is the object a vector?
[1] TRUE
> attributes(myvar) <- list(my_attr="foo")
> myvar
 [1]  1  2  3  4  5  6  7  8  9 10
attr(,"my_attr")
[1] "foo"
> is.vector(myvar)  # Is the object a vector?
[1] FALSE
> myvar <- 0
> attributes(myvar) <- list(class="Date")
> myvar  # "Date" object
[1] "1970-01-01"
> structure(0, class="Date")  # "Date" object
[1] "1970-01-01"
```

Modifying *class* Attributes

Objects without an explicit *class* don't have a *class* attribute, and the function class() returns the implicit *class*.

The *class* of an object can be modified using the "class() <-" assignment.

An object can have a main *class*, and also an inherited *class* (the *class* attribute can be a vector of strings).

The function unclass() removes the explicit class attribute from an object.

```
> myvar <- matrix(runif(10), 2, 5)
> class(myvar)  # Has implicit class
[1] "matrix" "array"
> # But no explicit "class" attribute
> attributes(myvar)
$dim
[1] 2 5
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "matrix"  "array"
> # Assign explicit "class" attribute
> class(myvar) <- "my_class"
> class(myvar)  # Has explicit "class"
[1] "my_class"
> # Has explicit "class" attribute
> attributes(myvar)
$dim
[1] 2 5

$class
[1] "my_class"
> is.matrix(myvar)  # Is the object a matrix?
[1] TRUE
> is.vector(myvar)  # Is the object a vector?
[1] FALSE
> attributes(unclass(myvar))
$dim
[1] 2 5
```

Implicit Class of Objects

If an object has no explicit *class*, then its implicit *class* is derived from its *mode* and `dim` attribute (except for `integer` vectors which have the implicit class `"integer"` derived from their *type*).

If an *atomic* object has a `dim` attribute, then its implicit *class* is `matrix` or `array`.

`Data frames` have no explicit `dim` attribute, but `dim()` returns a value, so they have an implicit `dim` attribute.

```
> # Integer implicit class derived from type
> myvar <- vector(mode="integer", length=10)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "integer"
> # Numeric implicit class derived from mode
> myvar <- vector(mode="numeric", length=10)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "numeric"
> # Adding dim attribute changes implicit class to matrix
> dim(myvar) <- c(5, 2)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "double"  "numeric" "matrix"  "array"
> # Data frames have implicit dim attribute
> myvar <- data.frame(aa=c("a", "b"), bb=1:2)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "list"        "list"        "data.frame"
> attributes(myvar)
$names
[1] "aa" "bb"

$class
[1] "data.frame"

$row.names
[1] 1 2
> dim(myvar)
[1] 2 2
```

Object Coercion

Coercion means changing the *type*, *mode*, or *class* of an object, often without changing the underlying data.

Changing the *mode* of an object can change its *class* as well, but not always.

Objects can be explicitly coerced using the "as.*" coercion functions.

Most coercion functions strip the *attributes* from the object.

Implicit coercion occurs when objects with different *modes* are combined into a vector, forcing the elements to have the same *mode*.

Implicit coercion can cause bugs that are difficult to trace.

The rule is that coercion is into larger types (`numeric` objects are coerced into `character` strings).

Coercion can introduce bad data, such as `NA` values.

```
> myvar <- 1:5
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "integer" "numeric" "integer"
> mode(myvar) <- "character"  # Coerce to "character"
> myvar
[1] "1" "2" "3" "4" "5"
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "character" "character" "character"
> # Explicitly coerce to "character"
> myvar <- as.character(1:5)
> c(typeof(myvar), mode(myvar), class(myvar))
[1] "character" "character" "character"
> matrixv <- matrix(1:10, 2, 5)  # Create matrix
> # Explicitly coerce to "character"
> matrixv <- as.character(matrixv)
> c(typeof(matrixv), mode(matrixv), class(matrixv))
[1] "character" "character" "character"
> # Coercion converted matrix to vector
> c(is.matrix(matrixv), is.vector(matrixv))
[1] FALSE  TRUE
> as.logical(0:3)  # Explicit coercion to "logical"
[1] FALSE  TRUE  TRUE  TRUE
> as.numeric(c(FALSE, TRUE, TRUE, TRUE))
[1] 0 1 1 1
> c(1:3, "a")  # Implicit coercion to "character"
[1] "1" "2" "3" "a"
> # Explicit coercion to "numeric"
> as.numeric(c(1:3, "a"))
[1]  1  2  3 NA
```

Basic R Objects

The quotation marks "" (or '') around a character string tell R that it's a string, not a variable name.

Vectors are the basic building blocks of R objects.

There are no scalars in R, and single values are stored as vectors of length "1".

A character string is also a vector with a single element, with the first element of the vector containing the string of text.

The colon binary operator ':' produces a vector.

The function c() combines objects into a vector.

The "[1]" symbol means the return value is a vector.

The function is.vector() returns TRUE if its argument is a vector, and returns FALSE otherwise.

```
> "Hello World!"  # Type some text
> # hello is a variable name, because it's not in quotes
> hello  # R interretsp "hello" as a variable name
> is.vector(1)  # Single number is a vector
> is.vector("a")  # String is a vector
> 4:8  # Create a vector
> # Create vector using c() combine function
> c(1, 2, 3, 4, 5)
> # Create vector using c() combine function
> c("a", "b", "c")
> # Create vector using c() combine function
> c(1, "b", "c")
```

Character Strings

Character `strings` are sequences of characters (and vectors of length one).

The function `nchar()` returns the length of a string.

Special characters in strings:
"\t" for TAB,
"\n" for new-line,
"\\" for a (single) backslash character

The function `cat()` concatenates strings and echos them to console, without returning any values.

The function `cat()` is useful in user-defined functions.

```
> str_var <- "Some string"
> str_var
[1] "Some string"
> str_var[1]
[1] "Some string"
> str_var[2]
[1] NA
>
> NROW(str_var)  # length of vector
[1] 1
> nchar(str_var)  # length of string
[1] 11
>
> # Concatenate and echo to console
> cat("Hello", "World!")
Hello World!
> cat("Enter\ttab")
Enter tab
> cat("Enter\nnewline")
Enter
newline
> cat("Enter\\backslash")
Enter\backslash
```

Manipulating Strings

The function `paste()` concatenates its arguments into a string, coerces them to characters if needed, and returns the string.

If a vector or list is passed to `paste()`, together with a `collapse` string, then `paste()` concatenates the elements into a string, separated by the `collapse` string.

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

Splitting on the "`.`" character requires surrounding it with brackets: "`[.]`", or using argument `fixed=TRUE`.

The function `substring()` extracts or replaces substrings in a character string.

The recycling rule extends the length to match the longest object.

```
> str_var1 <- "Hello"  # Define a character string
> str_var2 <- "World!"  # Define a character string
> paste(str_var1, str_var2, sep=" ")  # Concatenate and return value
[1] "Hello World!"
> cat(str_var1, str_var2)  # Concatenate and echo to console
Hello World!
> paste("a", 1:4, sep="-")  # Convert, recycle and concatenate
[1] "a-1" "a-2" "a-3" "a-4"
> paste(c("a1", "a2", "a3"), collapse="+")  # Collapse vector to str
[1] "a1+a2+a3"
> paste(list("a1", "a2", "a3"), collapse="+")
[1] "a1+a2+a3"
> paste("Today is", Sys.time())  # Coerce and concatenate strings
[1] "Today is 2022-09-05 17:13:53"
> paste("Today is", format(Sys.time(), "%B-%d-%Y"))
[1] "Today is September-05-2022"
> strsplit("Hello World", split="r")  # Split string
[[1]]
[1] "Hello Wo" "ld"
> strsplit("Hello.World", split="[.]")  # Split string
[[1]]
[1] "Hello" "World"
> strsplit("Hello.World", split=".", fixed=TRUE)  # Split string
[[1]]
[1] "Hello" "World"
> substring("Hello World", 3, 6)  # Extract characters from 3 to 6
[1] "llo "
```

Regular Expressions in R

R has Regex functions for pattern matching and replacement.

The function `gsub()` replaces all matches of a pattern in a string.

The function `grep()` searches for matches of a pattern in a string.

The function `glob2rx()` converts globbing wildcard patterns into regular expressions.

```
> gsub("is", "XX", "is this gratis?")  # Replace "is" with "XX"
[1] "XX thXX gratXX?"
>
> grep("b", c("abc", "xyz", "cba d", "bbb"))  # Get indexes
[1] 1 3 4
>
> grep("b", c("abc", "xyz", "cba d", "bbb"), value=TRUE)  # Get valu
[1] "abc"   "cba d" "bbb"
>
> glob2rx("abc.*")  # Convert globs into regex
[1] "^abc\\."
> glob2rx("*.doc")
[1] "^.*\\.doc$"
```

Vectors

Vectors are the basic building blocks of R objects.

There are no scalars in R, and single values are stored as vectors of length "1".

The function c() combines values into a vector.

The function is.vector() returns TRUE if its argument is a vector, and returns FALSE otherwise.

The object letters is a constant and a vector,

```
> is.vector(1)  # Single number is a vector
[1] TRUE
> is.vector("a")  # String is a vector
[1] TRUE
> vectorv <- c(8, 6, 5, 7)  # Create vector
> vectorv
[1] 8 6 5 7
> vectorv[2]  # Extract second element
[1] 6
> # Extract all elements, except the second element
> vectorv[-2]
[1] 8 5 7
> # Create Boolean vector
> c(FALSE, TRUE, TRUE)
[1] FALSE  TRUE  TRUE
> # Extract second and third elements
> vectorv[c(FALSE, TRUE, TRUE)]
[1] 6 5
> letters[5:10]  # Vector of letters
[1] "e" "f" "g" "h" "i" "j"
> c("a", letters[5:10])  # Combine two vectors of letters
[1] "a" "e" "f" "g" "h" "i" "j"
```

Creating Vectors

The colon operator (":") provides a simple way of creating a numeric vector.

The function vector() returns a vector of the specified *mode*.

The functions seq(), seq_len(), and seq_along() return a sequence (vector) of numbers.

The function rep() replicates an object multiple times.

The functions character() and numeric() return zero-length vectors of the specified *mode* (not to be confused with a NULL object).

Zero length vectors are not the same as NULL objects.

```
> 0:10  # Vector of integers from 0 to 10
 [1]  0  1  2  3  4  5  6  7  8  9 10
> vector()  # Create empty vector
logical(0)
> vector(mode="numeric", length=10)  # Numeric vector of zeros
 [1] 0 0 0 0 0 0 0 0 0 0
> seq(10)  # Sequence from 1 to 10
 [1]  1  2  3  4  5  6  7  8  9 10
> seq(along=(-5:5))  # Instead of 1:NROW(obj)
 [1]  1  2  3  4  5  6  7  8  9 10 11
> seq_along(c("a", "b", "c"))  # Instead of 1:NROW(obj)
[1] 1 2 3
> seq(from=0, to=1, len=11)  # Decimals from 0 to 1.0
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(from=0, to=1, by=0.1)  # Decimals from 0 to 1.0
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> seq(-2,2, len=11)  # 10 numbers from -2 to 2
 [1] -2.0 -1.6 -1.2 -0.8 -0.4  0.0  0.4  0.8  1.2  1.6  2.0
> rep(100, times=5)  # Replicate a number
[1] 100 100 100 100 100
> character(5)  # Create empty character vector
[1] "" "" "" "" ""
> numeric(5)  # Create empty numeric vector
[1] 0 0 0 0 0
> numeric(0)  # Create zero-length vector
numeric(0)
```

Arithmetic and Logical Operations on Vectors

Vectors can be multiplied and squared element by element, as if they were single elements.

When vectors are manipulated as if they were single elements, then R automatically performs a loop over the vector elements, and applies the operation element-wise.

This is a very powerful feature of R called *vectorized arithmetic*.

Vectorized arithmetic avoids writing loops and simplifies notation.

Vectors can be combined together and appended.

```
> 2*4:8  # Multiply a vector
> 2*(4:8)  # Multiply a vector
> 4:8/2  # Divide a vector
> (0:10)/10  # Divide vector - decimals from 0 to 1.0
> vectorv <- c(8, 6, 5, 7)  # Create vector
> vectorv
> # Boolean vector TRUE if element is equal to second one
> vectorv == vectorv[2]
> # Boolean vector TRUE for elements greater than six
> vectorv > 6
> 2*vectorv  # Multiply all elements by 2
> vectorv^2  # Square all elements
> c(11, 5:10)  # Combine two vectors
> c(vectorv, 2.0)  # Append number to vector
```

Naming and Manipulating Vectors

Vector elements can be assigned `namesv` using a list of symbol-value pairs.

The function `names()` returns the `namesv` attribute of an object.

The `namesv` attribute of a vector can be modified by assigning to the `names()` function ("names() <-" assignment).

The function `unname()` removes the `namesv` attribute.

The function `structure()` adds attributes (specified as symbol=value pairs) to an object, and returns it.

```
> vectorv <- # Create named vector
+  c(pi_const=pi, euler=exp(1), gamma=-digamma(1))
> vectorv
pi_const   euler    gamma
   3.142   2.718    0.577
> names(vectorv)  # Get names of elements
[1] "pi_const" "euler"     "gamma"
> vectorv["euler"]  # Get element named "euler"
euler
 2.72
> names(vectorv) <- c("pie","eulery","gammy")  # Rename elements
> vectorv
   pie eulery  gammy
 3.142  2.718  0.577
> unname(vectorv)  # Remove names attribute
[1] 3.142 2.718 0.577
> letters[5:10]  # Vector of letters
[1] "e" "f" "g" "h" "i" "j"
> c("a", letters[5:10])  # Combine two vectors of letters
[1] "a" "e" "f" "g" "h" "i" "j"
> # Create named vector
> structure(sample(1:5), names=paste0("el", 1:5))
el1 el2 el3 el4 el5
  1   2   5   4   3
```

Subsetting Vectors

Vector elements can be *subset* (indexed, referenced) using the "[]" operator.

Vectors can be *subset* using vectors of:

- positive integers,
- negative integers,
- characters (names),
- Boolean vectors,

Negative integers remove the vector elements.

Subsetting with *zero* returns a zero-length vector.

A named vector can be *subset* using element names.

```
> vectorv  # Named vector
  pie eulery  gammy
3.142  2.718  0.577
> # Extract second element
> vectorv[2]
eulery
 2.72
> # Extract all elements, except the second element
> vectorv[-2]
  pie gammy
3.142 0.577
> # Extract zero elements - returns zero-length vector
> vectorv[0]
named numeric(0)
> # Extract second and third elements
> vectorv[c(FALSE, TRUE, TRUE)]
eulery  gammy
 2.718  0.577
> # Extract elements using their names
> vectorv["eulery"]
eulery
 2.72
> # Extract elements using their names
> vectorv[c("pie", "gammy")]
  pie gammy
3.142 0.577
> # Subset whole vector
> vectorv[] <- 0
```

Filtering Vectors

Filtering means extracting elements from a vector that satisfy a logical condition.

When logical comparison operators are applied to vectors, they produce `Boolean` vectors.

`Boolean` vectors can then be applied to subset the original vectors, to extract their elements.

The function `which()` returns the indices of the `TRUE` elements of a `Boolean` vector or array.

```
> vectorv <- runif(5)
> vectorv
[1] 0.521 0.734 0.906 0.657 0.386
> vectorv > 0.5  # Boolean vector
[1]  TRUE  TRUE  TRUE  TRUE FALSE
> # Boolean vector of elements equal to the second one
> vectorv == vectorv[2]
[1] FALSE  TRUE FALSE FALSE FALSE
> # Extract all elements equal to the second one
> vectorv[vectorv == vectorv[2]]
[1] 0.734
> vectorv < 1  # Boolean vector of elements less than one
[1] TRUE TRUE TRUE TRUE TRUE
> # Extract all elements greater than one
> vectorv[vectorv > 1]
numeric(0)
> vectorv[vectorv > 0.5]  # Filter elements > 0.5
[1] 0.521 0.734 0.906 0.657
> which(vectorv > 0.5)  # Index of elements > 0.5
[1] 1 2 3 4
```

Factors

Factors are similar to vectors, but their elements can only take values from a set of *levels*.

Factors are designed for categorical data which can only take certain values.

The function `factor()` converts a vector into a factor.

Factors have two attributes: *class* (equal to `"factor"`) and `levels` (the allowed values).

Although factors aren't vectors, the data underlying a factor is an integer vector, called an *encoding vector*.

The function `as.numeric()` extracts the encoding vector (indices) of a factor.

The function `as.vector()` coerces a factor to a character vector.

```
> # Create factor vector
> factorv <- factor(c("b", "c", "d", "a", "c", "b"))
> factorv
[1] b c d a c b
Levels: a b c d
> factorv[3]
[1] d
Levels: a b c d
> # Get factor attributes
> attributes(factorv)
$levels
[1] "a" "b" "c" "d"

$class
[1] "factor"
> # Get allowed values
> levels(factorv)
[1] "a" "b" "c" "d"
> # Get encoding vector
> as.numeric(factorv)
[1] 2 3 4 1 3 2
> is.vector(factorv)
[1] FALSE
> # Coerce vector to factor
> as.factor(1:5)
[1] 1 2 3 4 5
Levels: 1 2 3 4 5
> # Coerce factor to character vector
> as.vector(as.factor(1:5))
[1] "1" "2" "3" "4" "5"
```

Tables of Categorical Data

The function `unique()` calculates the unique elements of an object.

The function `levels()` extracts the `levels` attribute of a factor (the allowed values).

A contingency table is a matrix that contains the frequency distribution of variables (factors) contained in a set of data.

The function `table()` calculates the frequency distribution of categorical data.

`sapply()` applies a function to a vector or a list of objects and returns a vector or a list.

```
> factorv
[1] b c d a c b
Levels: a b c d
> # Get unique elements
> unique(factorv)
[1] b c d a
Levels: a b c d
> # Get levels attribute of the factor
> levels(factorv)
[1] "a" "b" "c" "d"
> # Calculate the factor elements from its levels
> levels(factorv)[as.numeric(factorv)]
[1] "b" "c" "d" "a" "c" "b"
> # Get contingency (frequency) table
> table(factorv)
factorv
a b c d
1 2 2 1
```

Classifying Continuous Numeric Data Into Categories

Numeric data that represents a *magnitude*, *intensity*, or *score* can be classified into categorical data, given a vector of *breakpoints*.

The *breakpoints* create intervals that correspond to different *categories*.

The *categories* combine elements that have a similar numeric *magnitude*.

`findInterval()` returns the indices of the intervals specified by "vec" that contain the elements of "x".

If there's an exact match, then `findInterval()` returns the same index as function `match()`.

If there's no exact match, then `findInterval()` finds the element of "vec" that is closest to, but not greater than, the element of "x".

If all the elements of "vec" are greater than the element of "x", then `findInterval()` returns zero.

`args()` displays the formal arguments of a function.

```
> # Display the formal arguments of findInterval
> args(findInterval)
function (x, vec, rightmost.closed = FALSE, all.inside = FALSE,
    left.open = FALSE)
NULL
> # Get index of the element of "vec" that matches 5
> findInterval(x=5, vec=c(3, 5, 7))
[1] 2
> match(5, c(3, 5, 7))
[1] 2
> # No exact match
> findInterval(x=6, vec=c(3, 5, 7))
[1] 2
> match(6, c(3, 5, 7))
[1] NA
> # Indices of "vec" that match elements of "x"
> findInterval(x=1:8, vec=c(3, 5, 7))
[1] 0 0 1 1 2 2 3 3
> # Return only indices of inside intervals
> findInterval(x=1:8, vec=c(3, 5, 7), all.inside=TRUE)
[1] 1 1 1 1 2 2 2 2
> # make rightmost interval inclusive
> findInterval(x=1:8, vec=c(3, 5, 7), rightmost.closed=TRUE)
[1] 0 0 1 1 2 2 2 3
```

Classifying Numeric Data Into Categories Example

Temperature can be categorized into "cold", "warm", "hot", etc.

A named numeric vector of *breakpoints* can be used to convert a temperature into one of the *categories*.

Breakpoints correspond to *categories* of the data.

The first *breakpoint* should correspond to the lowest *category*, and should have a value less than any of the data.

```
> # Named numeric vector of breakpoints
> brea_ks <- c(freezing=0, very_cold=30, cold=50,
+       pleasant=60, warm=80, hot=90)
> brea_ks
 freezing very_cold      cold  pleasant      warm      hot
        0        30        50        60        80        90
> tempe_ratures <- runif(10, min=10, max=100)
> feels_like <- names(
+   brea_ks[findInterval(x=tempe_ratures, vec=brea_ks)])
> names(tempe_ratures) <- feels_like
> tempe_ratures
     cold very_cold  pleasant  freezing  pleasant  freezing  pleasan
     54.9      31.0      76.2      15.6      69.1      13.6      65.
     cold  pleasant
     50.7      72.2
```

Converting Numeric Data Into Factors Using cut()

The function cut() converts a numeric vector into a vector of factors, representing the intervals to which the numeric values belong.

cut() divides the range of values into intervals, based on a vector of breaks.

cut() then assigns factors to the numeric values, representing the intervals to which the numeric values belong.

The parameter breaks is a numeric vector of break points that divide the range of values into intervals.

The argument "labels" is a vector of labels for the intervals.

The argument "right" is a Boolean indicating if the intervals should be closed on the right (and open on the left), or vice versa.

cut() can produce the same classification as findInterval(), but findInterval() is faster than cut(), because it's a compiled function.

```
> datav <- sample(0:6) + 0.1
> datav
[1] 0.1 4.1 5.1 2.1 1.1 6.1 3.1
> cut(x=datav, breaks=c(2, 4, 6, 8))
[1] <NA>  (4,6] (4,6] (2,4] <NA>  (6,8] (2,4]
Levels: (2,4] (4,6] (6,8]
> rbind(datav, cut(x=datav, breaks=c(2, 4, 6, 8)))
       [,1] [,2] [,3] [,4] [,5] [,6] [,7]
datav  0.1  4.1  5.1  2.1  1.1  6.1  3.1
         NA  2.0  2.0  1.0   NA  3.0  1.0
> # cut() replicates findInterval()
> cut(x=1:8, breaks=c(3, 5, 7), labels=1:2,
+     right=FALSE)
[1] <NA> <NA> 1    1    2    2    <NA> <NA>
Levels: 1 2
> findInterval(x=1:8, vec=c(3, 5, 7))
[1] 0 0 1 1 2 2 3 3
> # findInterval() is a compiled function, so it's faster than cut()
> vectorv <- rnorm(1000)
> summary(microbenchmark(
+   find_interval=
+     findInterval(x=vectorv, vec=c(3, 5, 7)),
+   cuut=
+     cut(x=vectorv, breaks=c(3, 5, 7)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
            expr  mean median
1 find_interval  4.97   4.14
2          cuut 57.48  47.68
```

Plotting Histograms of Frequency Data

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

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

If the argument `freq` is TRUE then the frequencies (counts) are plotted, and if it's FALSE then the probability density is plotted (with total area equal to 1).

The function `density()` calculates a kernel estimate of the probability density for a sample of data.

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

VTI Return Distribution

```
> # Calculate VTI percentage returns
> returns <- na.omit(rutils::etfenv$returns$VTI)
> # Plot histogram
> x11(width=6, height=5)
> par(mar=c(1, 1, 1, 1), oma=c(2, 2, 2, 0))
> madv <- mad(returns)
> histp <- hist(returns, breaks=100,
+    main="", xlim=c(-5*madv, 5*madv),
+    xlab="", ylab="", freq=FALSE)
```
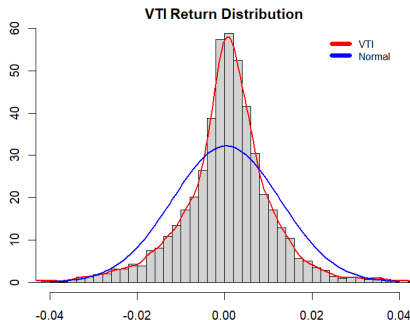
```
> # Draw kernel density of histogram
> lines(density(returns), col="red", lwd=2)
> # Add density of normal distribution
> curve(expr=dnorm(x, mean=mean(returns), sd=sd(returns)),
+ add=TRUE, type="l", lwd=2, col="blue")
> title(main="VTI Return Distribution", line=0)
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+    leg=c("VTI", "Normal"), bty="n",
+    lwd=6, bg="white", col=c("red", "blue"))
> # Total area under histogram
> sum(diff(histp$breaks) * histp$density)
```

Matrices

The function `matrix()` creates a matrix from a vector, and the matrix dimensions.

By default `matrix()` creates matrices column-wise, unless the argument `byrow=TRUE` is used.

The elements of matrices can be subset (referenced) using the "`[]`" operator.

The functions `nrow()` and `ncol()` return the number of rows and columns of a matrix.

The functions `NROW()` or `NCOL()` also return the number of rows or columns of a matrix, but they can also be applied to vectors, and treat vectors as single column matrices.

```
> matrixv <- matrix(5:10, nrow=2, ncol=3)  # Create a matrix
> matrixv  # By default matrices are constructed column-wise
     [,1] [,2] [,3]
[1,]    5    7    9
[2,]    6    8   10
> # Create a matrix row-wise
> matrix(5:10, nrow=2, byrow=TRUE)
     [,1] [,2] [,3]
[1,]    5    6    7
[2,]    8    9   10
> matrixv[2, 3]  # Extract third element from second row
[1] 10
> matrixv[2, ]  # Extract second row
[1] 6  8 10
> matrixv[, 3]  # Extract third column
[1] 9 10
> matrixv[, c(1,3)]  # Extract first and third column
     [,1] [,2]
[1,]    5    9
[2,]    6   10
> matrixv[, -2]  # Remove second column
     [,1] [,2]
[1,]    5    9
[2,]    6   10
> # Subset whole matrix
> matrixv[] <- 0
> # Get the number of rows or columns
> nrow(vectorv); ncol(vectorv)
NULL
NULL
> NROW(vectorv); NCOL(vectorv)
[1] 1000
[1] 1
> nrow(matrixv); ncol(matrixv)
[1] 2
[1] 3
> NROW(matrixv); NCOL(matrixv)
```

Matrix Attributes

Arrays are vectors with a dimension attribute.

Matrices are two-dimensional arrays.

The dimension attribute of a matrix is an integer vector of length 2 (nrow, ncol).

The `dimnames` attribute is a list, with vector elements containing row and column names.

A named matrix can be subset using row and column names.

```
> attributes(matrixv)  # Get matrix attributes
$dim
[1] 2 3
> dim(matrixv)  # Get dimension attribute
[1] 2 3
> class(matrixv)  # Get class attribute
[1] "matrix" "array"
> rownames(matrixv) <- c("row1", "row2")  # Rownames attribute
> colnames(matrixv) <- c("col1", "col2", "col3")  # Colnames attrib
> matrixv
     col1 col2 col3
row1    0    0    0
row2    0    0    0
> matrixv["row2", "col3"]  # Third element from second row
[1] 0
> names(matrixv)  # Get the names attribute
NULL
> dimnames(matrixv)  # Get dimnames attribute
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"
> attributes(matrixv)  # Get matrix attributes
$dim
[1] 2 3

$dimnames
$dimnames[[1]]
[1] "row1" "row2"

$dimnames[[2]]
[1] "col1" "col2" "col3"
```

Matrix Subsetting

Matrices can be subset in a similar way as Vectors, either by indices (integers), by characters (names), or `Boolean` vectors.

Subsetting a matrix to a single row or column produces a vector, unless the parameter "drop=FALSE" is used.

Subsetting with the parameter "drop=FALSE" prevents the implicit coercion and preserves the matrix *class*.

This is an example of implicit coercion in R, which can cause difficult to trace bugs.

```
> matrixv  # matrix with column names
     col1 col2 col3
row1    0    0    0
row2    0    0    0
> matrixv[1, ]  # Subset rows by index
col1 col2 col3
   0    0    0
> matrixv[, "col1"]  # Subset columns by name
row1 row2
   0    0
> matrixv[, c(TRUE, FALSE, TRUE)]  # Subset columns Boolean vector
     col1 col3
row1    0    0
row2    0    0
> matrixv[1, ]  # Subsetting can produce a vector!
col1 col2 col3
   0    0    0
> class(matrixv); class(matrixv[1, ])
[1] "matrix" "array"
[1] "numeric"
> is.matrix(matrixv[1, ]); is.vector(matrixv[1, ])
[1] FALSE
[1] TRUE
> matrixv[1, , drop=FALSE]  # Drop=FALSE preserves matrix
     col1 col2 col3
row1    0    0    0
> class(matrixv[1, , drop=FALSE])
[1] "matrix" "array"
> is.matrix(matrixv[1, , drop=FALSE]); is.vector(matrixv[1, , drop=F
[1] TRUE
[1] FALSE
```

Logical Operators

R has the following logical operators:

- "<" less than,
- "<=" less than or equal to,
- ">" greater than,
- ">=" greater than or equal to,
- "==" exactly equal to,
- "!=" not equal to,
- "!x" Not x,
- "x & y" x AND y,
- "x | y" x OR y,

These operators are applied to vectors element-wise.

```
> TRUE | FALSE
> TRUE | NA
> vector1 <- c(2, 4, 6)
> vector1 < 5  # Element-wise comparison
> (vector1 < 5) & (vector1 > 3)
> vector1[(vector1 < 5) & (vector1 > 3)]
> vector2 <- c(-10, 0, 10)
> vector1 < vector2
> c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)
> c(FALSE, TRUE, FALSE) | c(TRUE, TRUE, FALSE)
```

Long Form Logical Operators

R also has two long form logical operators:

- "x && y" x AND y,
- "x || y" x OR y,

These operators differ from the short form operators in two ways:

- They only evaluate the first elements of their vector arguments,
- They short-circuit (stop evaluation as soon as the expression is determined),

Rule of Thumb

- Use "&&" and "||" in if-clauses,

```
> c(FALSE, TRUE, FALSE) && c(TRUE, TRUE, FALSE)
> c(FALSE, TRUE, FALSE) || c(TRUE, TRUE, FALSE)
> echo_true <- function() {cat("echo_true\t"); TRUE}
> echo_false <- function() {cat("echo_false\t"); FALSE}
> echo_true() | echo_false()
> echo_true() || echo_false()  # echo_false isn't evaluated at all
> vectorv <- c(2, 4, 6)
> # Works (does nothing) using '&&'
> if (is.matrix(vectorv) && (vectorv[2, 3] > 0)) {
+     vectorv[2, 3] <- 1
+ }
> # No short-circuit so fails (produces an error)
> if (is.matrix(vectorv) & (vectorv[2, 3] > 0)) {
+     vectorv[2, 3] <- 1
+ }
```

Arithmetic Operators

Arithmetic *operators* perform arithmetic operations on numeric or complex vectors,

- "+" performs addition,

- "−" performs subtraction,

- "∗" performs multiplication,

- "/" performs division,

- "^" and "∗∗" perform exponentiation,

```
> ?Arithmetic
> 4.7 * 0.5  # Multiplication
> 4.7 / 0.5  # division
> # Exponentiation
> 2**3
> 2^3
```

Comparing Objects With identical() and all.equal()

The function identical() tests if two objects are exactly the same, and always returns a single logical TRUE or FALSE (never NA or logical vectors).

For atomic arguments identical() often gives the same result as the "==" operator, but it's not synonymous with it in general.

The "==" operator applies the *recycling rule* to vector arguments and returns logical vectors, but identical() doesn't and returns a single logical value.

The function all.equal() tests the equality of two objects to within the square root of the *machine precision*.

The variable .Machine contains information about the numerical characteristics of the computer R is running on, such as the largest double and integer numbers, and the *machine precision*.

```
> numv <- 2
> numv==2
> identical(numv, 2)
>
> identical(numv, NULL)
> # This doesn't work:
> # numv==NULL
> is.null(numv)
>
> vectorv <- c(2, 4, 6)
> vectorv==2
> identical(vectorv, 2)
>
> # numv is equal to "1.0" within machine precision
> numv <- 1.0 + 2*sqrt(.Machine$double.eps)
> all.equal(numv, 1.0)
>
> # Info machine precision of computer R is running on
> # ?.Machine
> # Machine precision
> .Machine$double.eps
```

Lookup and Matching Using `which()` and `match()`

The function `which()` returns the indices of the `TRUE` elements of a Boolean vector or array.

If the argument is an `array` and `arr.ind=TRUE`, then `which()` returns a matrix with rows containing the indices of the `TRUE` elements.

The functions `which.max()` and `which.min()` return the index of the minimum or maximum of a `numeric` or `Boolean` vector.

`match()` returns the index of the vector element that *exactly* matches its first argument.

If it doesn't find an exact match then it returns `NA`.

The expressions `match(x, vec_tor)` and `min(which(vec_tor == x))` produce the same result, but `match()` can be faster for large vectors.

```
> vectorv <- sample(1e3, 1e3)
> matrixv <- matrix(vectorv, ncol=4)
> which(vectorv == 5)
> match(5, vectorv)
> # Equivalent but slower than above
> (1:NROW(vectorv))[vectorv == 5]
> which(vectorv < 5)
> # Find indices of TRUE elements of Boolean matrix
> which((matrixv == 5)|(matrixv == 6), arr.ind=TRUE)
> # Equivalent but slower than above
> arrayInd(which((matrixv == 5)|(matrixv == 6)),
+     dim(matrixv), dimnames(matrixv))
> # Find index of largest element
> which.max(vectorv)
> which(vectorv == max(vectorv))
> # Find index of smallest element
> which.min(vectorv)
> # Benchmark match() versus which()
> all.equal(match(5, vectorv),
+     min(which(vectorv == 5)))
> library(microbenchmark)
> summary(microbenchmark(
+     match=match(5, vectorv),
+     which=min(which(vectorv == 5)),
+     times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Lookup and Matching Using `%in%` and `any()`

The binary operator `%in%` returns a Boolean vector with `TRUE` values corresponding to elements that have matches.

`%in%` is a wrapper for `match()` defined as follows:
`"%in%" <- function(x, table) match(x, table, nomatch=0) > 0.`

`%in%` never returns NA, so it's preferred in `if()` statements.

`any()` returns `TRUE` if at least one element of a Boolean vector is `TRUE`, and `FALSE` otherwise.

The function `pmatch()` performs partial matching of strings.

```
> # Does 5 belong in vectorv?
> 5 %in% vectorv
> match(5, vectorv, nomatch=0) > 0
> # Does (-5) belong in vectorv?
> (-5) %in% vectorv
> c(5, -5) %in% vectorv
> match(-5, vectorv)
> # Equivalent to "5 %in% vectorv"
> any(vectorv == 5)
> # Equivalent to "(-5) %in% vectorv"
> any(vectorv == (-5))
> # Any negative values in vectorv?
> any(vectorv < 0)
> # Example of use in if() statement
> if (any(vectorv < 2))
+   cat("vector contains small values\n")
> # Partial matching of strings
> pmatch("med", c("mean", "median", "mode"))
```

Finding Closest Match Using `findInterval()`

The function `match()` returns the index of the vector element that *exactly* matches its first argument.

If `match()` doesn't find an exact match then it returns `NA`.

The function `findInterval()` returns the indices of the intervals specified by `"vec"` that contain the elements of `"x"`.

If there's an exact match, then `findInterval()` returns the same index as function `match()`.

If there's no exact match, then `findInterval()` finds the element of `"vec"` that is closest to, but not greater than, the element of `"x"`.

```
> str(findInterval)
> # Get index of the element of "vec" that matches 5
> findInterval(x=5, vec=c(3, 5, 7))
> match(5, c(3, 5, 7))
> # No exact match
> findInterval(x=6, vec=c(3, 5, 7))
> match(6, c(3, 5, 7))
> # Indices of "vec" that match elements of "x"
> findInterval(x=1:8, vec=c(3, 5, 7))
> # Return only indices of inside intervals
> findInterval(x=1:8, vec=c(3, 5, 7),
+        all.inside=TRUE)
> # Make rightmost interval inclusive
> findInterval(x=1:8, vec=c(3, 5, 7),
+        rightmost.closed=TRUE)
```

Assignment Operators

The standard assignment operator in R is "<-".

Both "<-" and "=" are valid assignment operators in R.

The "<-" operator may cause an error if R confuses it with the "<" logical operator.

But they differ in *scope* and *precedence* ("<-" has higher precedence than "=").

The "=" operator is used for named arguments in function calls.

When variables are assigned within an argument list using the "=" operator, their *scope* is limited to the function.

Rule of Thumb:
Use "<-" in R scripts and inside functions,
Use "=" only in function calls.

```
> numv1 <- 3   # "<-" and "=" are valid assignment operators
> numv1
> numv1 = 3
> numv1
> 2<-3  # "<" operator confused with "<-"
> 2 < -3  # Add space or brackets to avoid confusion
> # "=" assignment within argument list
> median(x=1:10)
> x  # x doesn't exist outside the function
> # "<-" assignment within argument list
> median(x <- 1:10)
> x  # x exists outside the function
```

The assign() Function

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

assign() can be used to either assign values to existing variables, or to create new variables.

assign() looks for the object name in the specified *environment*, and assigns a value to it.

If assign() can't find the object name, then it creates it.

assign() expects a character string as its argument.

If a object name is passsed to assign(), then it evaluates that object to get the string it contains.

If the object doesn't contain a string, then assign() produces an error.

```
> myvar <- 1  # Create new object
> assign(x="myvar", value=2)  # Assign value to existing object
> myvar
> rm(myvar)  # Remove myvar
> assign(x="myvar", value=3)  # Create new object from name
> myvar
> # Create new object in new environment
> new_env <- new.env()  # Create new environment
> assign("myvar", 3, envir=new_env)  # Assign value to name
> ls(new_env)  # List objects in "new_env"
> new_env$myvar
> rm(list=ls())  # delete all objects
> symbol <- "myvar"  # define symbol containing string "myvar"
> assign(symbol, 1)  # Assign value to "myvar"
> ls()
> myvar
> assign("symbol", "new_var")
> assign(symbol, 1)  # Assign value to "new_var"
> ls()
> symbol <- 10
> assign(symbol, 1)  # Can't assign to non-string
```

Applying `assign()` to Lists of Names

`assign()` allows creating new objects from listv or vectors of names (character strings), such as column names.

```
> rm(list=ls())  # delete all objects
> # Create individual vectors from column names of EuStockMarkets
> for (colname in colnames(EuStockMarkets)) {
+ # Assign column values to column names
+   assign(colname, EuStockMarkets[, colname])
+ }  # end for
> ls()
> head(DAX)
> head(EuStockMarkets[, "DAX"])
> identical(DAX, EuStockMarkets[, "DAX"])
```

Retrieving Objects Using get()

The function get() accepts a character string and returns the value of the corresponding object in a specified *environment*.

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

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.

```
> # Create new environment
> test_env <- new.env()
> # Pass string as name to create new object
> assign("myvar1", 2, envir=test_env)
> # Create new object using $ string referencing
> test_env$myvar2 <- 1
> # List objects in new environment
> ls(test_env)
> # Reference an object by name
> test_env$myvar1
> # Reference an object by string name using get
> get("myvar1", envir=test_env)
> # Retrieve and assign value to object
> assign("myvar1",
+        2*get("myvar1", envir=test_env),
+        envir=test_env)
> get("myvar1", envir=test_env)
> # Return all objects in an environment
> mget(ls(test_env), envir=test_env)
> # delete environment
> rm(test_env)
```

The Parenthesis "()" and Curly Braces "{}" Operators

The parenthesis "()" and curly braces "{}" operators are used to enclose and to group (combine) expressions.

The parenthesis "()" and curly braces "{}" operators are functions, and they return values.

An expression enclosed by the parenthesis "()" operator is evaluated separatately from other expressions, and its result is returned.

Enclosing expressions in parenthesis makes them less ambiguous.

The curly braces "{}" operator can group several expressions, that can be written either on separate lines, or be separated by the semicolon ";" operator.

The curly braces "{}" operator returns the last expression it encloses.

Both the parenthesis "()" and curly braces "{}" operators are functions, and executing them requires a little additional processing time.

The square braces (brackets) "[]" operator subsets (references) the elements of vectors, matrices, and listv.

```
> # expressions enclosed in parenthesis are less ambiguous
> -2:5
> (-2):5
> -(2:5)
> # expressions enclosed in parenthesis are less ambiguous
> -2*3+5
> -2*(3+5)
>
> # expressions can be separated by semicolons or by lines
> {1+2; 2*3; 1:5}
> # or
> {1+2
+ 2*3
+ 1:5}
>
> matrixv <- matrix(nr=3, nc=4)
> matrixv <- 0
> # subset whole matrix
> matrixv[] <- 0
>
> # parenthesis and braces require a little additional processing ti
> library(microbenchmark)
> summary(microbenchmark(
+   basep=sqrt(rnorm(10000)^2),
+   parven=sqrt(((((rnorm(10000)^2))))),
+   bra_ce=sqrt({{{{rnorm(10000)^2}}}}),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

The "if () else" Control Statement

R has the familiar "if () {...} else {...}" statement to control execution flow depending on logical conditions.

The logical conditions must be either a Boolean or numeric type, otherwise an error is produced.

The "else" statement can also be omitted.

"if" statements can be nested using multiple "else if" statements.

```
> numv1 <- 1
>
> if (numv1) {  # numeric zero is FALSE, all other numbers are TRUE
+    numv2 <- 4
+ } else if (numv1 == 0) {  # 'else if' together on same line
+    numv2 <- 0
+ } else {  # 'else' together with curly braces
+    numv2 <- -4
+ }  # end if
>
> numv2
```

The switch() Control Statement

The function switch() matches its first argument "EXPR" with one of the symbols in the following arguments, evaluates the corresponding expression, and returns it.

The arguments that follow the first argument "EXPR" should be given as *symbol=value* pairs.

If "EXPR" is a character string, then the expression bound to that symbol is returned by switch().

If "EXPR" is an integer, then switch() returns the expression from that position.

If switch() can't match "EXPR" to any symbol, then it returns NULL invisibly.

Using switch() is a convenient alternative to a cascade of "if () else" statements.

The function match.arg() matches a string to one of the possible values, and returns the matched value, or produces an error if it can't match it.

```
> switch("a", a="aaahh", b="bee", c="see", d=2,
+       else this")
> switch("c", a="aaahh", b="bee", c="see", d=2,
+       else this")
> switch(3, a="aaahh", b="bee", c="see", d=2,
+       else this")
> switch("cc", a="aaahh", b="bee", c="see", d=2,
+       else this")
> # measure of central tendency
> centra_lity <- function(input,
+     method=c("mean", "mean_narm", "median")) {
+ # validate "method" argument
+   method <- match.arg(method)
+   switch(method,
+ mean=mean(input),
+ mean_narm=mean(input, na.rm=TRUE),
+ median=median(input))
+ }  # end centra_lity
> myvar <- rnorm(100, mean=2)
> centra_lity(myvar, "mean")
> centra_lity(myvar, "mean_narm")
> centra_lity(myvar, "median")
```

Iteration Using `for()` and `while()` Loops

The `for()` loop statement:

```
> for (indeks in vectorv) {ex_pressions}
```

iterates the *dummy* variable in_dex over the elements of the vector or list vec_tor, and evaluates in a loop the ex_pressions contained in the body of the `for()` loop.

Upon loop exit the *dummy* variable in_dex is left equal to the last element of the vector vec_tor.

`while()` loops start by testing their logical condition, and they repeat executing the loop body until that condition is `FALSE`.

But `while()` loops risk producing infinite loops if not written properly, so Use Them With Care!

```
> color_list <- list("red", "white", "blue")
> # loop over list
> for (some_color in color_list) {
+   print(some_color)
+ }  # end for
> # loop over vector
> for (indeks in 1:3) {
+   print(color_list[[indeks]])
+ }  # end for
>
> # while loops require initialization
> indeks <- 1
> # while loop
> while (indeks < 4) {
+   print(color_list[[indeks]])
+   indeks <- indeks + 1
+ }  # end while
```

Performing Loops Using `for()` and `apply()`

The `for()` loop doesn't return a value, so values calculated in the `for()` loop body must be assigned to variables in the parent environment, or otherwise they are lost.

The expressions in the `for()` loop body have access to variables in the parent environment in which the `for()` loop is executed, and they can modify those variables.

So even though `for()` loops don't return a value, they can be used to perform calculations on variables in the parent environment, but this is discouraged since it can produce errors that are hard to debug.

Rule of Thumb:

- `for()` loops are preferred for producing *side effects*, like plotting or reading and writing data to files,

- `apply()` loops are preferred for performing calculations which produce vectors or matrices of values,

```
> # loop over a vector and overwrite it
> vectorv <- integer(7)
> for (i in 1:7) {
+   cat("Changing element:", i, "\n")
+   vectorv[i] <- i^2
+ }  # end for
> # equivalent way (without cat side effect)
> for (i in seq_along(vectorv))
+   vectorv[i] <- i^2
>
> # sapply() loop returns vector of values
> vectorv <- sapply(seq_along(vectorv),
+           function(x) (x^2))
```

Fibonacci Sequence Using `for()` Loop

The *Fibonacci* sequence of integers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$
$$F1 = 0, F2 = 1,$$
$$F_n = 0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

The *Fibonacci* sequence was invented by the *Indian* mathematician Virahanka in the 8th century AD, and later described by the Italian mathematician *Fibonacci* in his famous treatise *Liber Abaci*.

Very often variables are initialized to `NULL` before the start of iteration.

A more efficient way to perform iteration is by pre-allocating the vector.

The function `numeric()` returns an zero length numeric vector.

The function `numeric(k)` returns a numeric vector of zeros of length k.

```
> # fib_seq <- numeric()  # zero length numeric vector
> # pre-allocate vector instead of "growing" it
> fib_seq <- numeric(10)
> fib_seq[1] <- 0  # initialize
> fib_seq[2] <- 1  # initialize
> for (i in 3:10) {  # perform recurrence loop
+   fib_seq[i] <- fib_seq[i-1] + fib_seq[i-2]
+ }  # end for
> fib_seq
```

Allocating Memory to Vectors and Matrices

R automatically allocates memory to new objects as needed during runtime, but at the cost of slowing down calculations.

Allocating memory of the correct *mode* speeds up calculations by avoiding automatic memory allocation by R.

The functions character(), integer(), and numeric() return zero-length vectors of the specified *mode*.

Zero length vectors are not the same as NULL objects.

The function character(k) returns a character vector of empty strings of length k.

The function integer(k) returns a integer vector of zeros of length k.

The function numeric(k) returns a numeric vector of zeros of length k.

The function vector() by default returns a Boolean vector, unless the *mode* is specified.

The function matrix() by default returns a Boolean matrix containing NA values, unless the *mode* is specified.

```
> # Allocate character vector
> character()
> character(5)
> is.character(character(5))
> # Allocate integer vector
> integer()
> integer(5)
> is.integer(integer(5))
> is.numeric(integer(5))
> # Allocate numeric vector
> numeric()
> numeric(5)
> is.integer(numeric(5))
> is.numeric(numeric(5))
> # Allocate Boolean vector
> vector()
> vector(length=5)
> # Allocate numeric vector
> vector(length=5, mode="numeric")
> is.null(vector())
> # Allocate Boolean matrix
> matrix()
> is.null(matrix())
> # Allocate integer matrix
> matrix(NA_integer_, nrow=3, ncol=2)
> is.integer(matrix(NA_integer_, nrow=3, ncol=2))
> # Allocate numeric matrix
> matrix(NA_real_, nrow=3, ncol=2)
> is.numeric(matrix(NA_real_, nrow=3, ncol=2))
```

Logical Operators Applied to Vectors and Matrices

When logical operators are applied to vectors and matrices, they are applied element-wise, producing `Boolean` vectors and matrices.

```
> vectorv <- sample(1:9)
> vectorv
> vectorv < 5  # Element-wise comparison
> vectorv == 5  # Element-wise comparison
> matrixv <- matrix(vectorv, ncol=3)
> matrixv
> matrixv < 5  # Element-wise comparison
> matrixv == 5  # Element-wise comparison
```

Coercing Vectors Into Matrices

Vectors can be coerced into matrices by adding a dimension attribute.

The `dimnames` attribute can be assigned a named list to convert it into a named matrix.

The function `structure()` adds attributes (specified as symbol=value pairs) to an object, and returns it.

```
> matrixv <- 1:6  # Create a vector
> class(matrixv)  # Get its class
> # Is it vector or matrix?
> c(is.vector(matrixv), is.matrix(matrixv))
> structure(matrixv, dim=c(2, 3))  # Matrix object
> # Adding dimension attribute coerces into matrix
> dim(matrixv) <- c(2, 3)
> class(matrixv)  # Get its class
> # Is it vector or matrix?
> c(is.vector(matrixv), is.matrix(matrixv))
> # Assign dimnames attribute
> dimnames(matrixv) <- list(rows=c("row1", "row2"),
+              columns=c("col1", "col2", "col3"))
> matrixv
```

Coercing Matrices Into Other Types

Matrices can be explicitly coerced using the `"as.*"` coercion functions.

But coercion functions strip the *attributes* from an object.

```
> matrixv <- matrix(1:10, 2, 5)  # Create matrix
> matrixv
> # as.numeric strips dim attribute from matrix
> as.numeric(matrixv)
> # Explicitly coerce to "character"
> matrixv <- as.character(matrixv)
> c(typeof(matrixv), mode(matrixv), class(matrixv))
> # Coercion converted matrix to vector
> c(is.matrix(matrixv), is.vector(matrixv))
```

Binding Vectors and Matrices Together

Vectors can be bound into matrices using the functions `cbind()` and `rbind()`.

The *recycling rule* allows operations on vectors of different lengths:

1. Vectors are scanned from left to right,

2. Shorter vectors are extended in length by recycling their values until they match the length of longer vectors,

```
> vector1 <- 1:3  # Define vector
> vector2 <- 6:4  # Define vector
> # Bind vectors into columns
> cbind(vector1, vector2)
> # Bind vectors into rows
> rbind(vector1, vector2)
> # Extend to four elements
> vector2 <- c(vector2, 7)
> # Recycling rule applied
> cbind(vector1, vector2)
> # Another example of recycling rule
> 1:6 + c(10, 20)
```

Replicating Objects Using `rep()`

The function `rep()` replicates vectors and listv a given number of times.

`rep()` accepts a vector or list "x", and an integer specifying the type and number of replications.

Argument "times" replicates the whole vector a given number of times.

Argument "each" replicates each vector element a given number of times.

Argument "length.out" replicates the whole vector a certain number of times, so that the output vector length is equal to "length.out".

```
> # Replicate a single element
> rep("a", 5)
> # Replicate the whole vector several times
> rep(c("a", "b"), 5)
> rep(c("a", "b"), times=5)
> # Replicate the first element, then the second, etc.
> rep(c("a", "b"), each=5)
> # Replicate to specified length
> rep(c("a", "b"), length.out=5)
```

Multiplying Vectors and Matrices

The multiplication "*" *operator* performs *element-wise* (*element-by-element*) multiplication of vectors and matrices.

By default the matrix elements are multiplied column-wise by the vector elements: the first matrix element in the first column is multiplied by the first vector element, then the second matrix column is multiplied by the remaining vector elements, etc.

The *recycling rule* is applied to the vector elements as needed.

The transpose function t() can be applied if we want to perform row-wise multiplication.

But the transpose function t() is very slow for large matrices.

A better choice is to use functions lapply() and do.call().

```
> # Define vector and matrix
> vector1 <- c(2, 4, 3)
> matrixv <- matrix(sample(1:12), ncol=3)
> # Multiply columns of matrix by vector
> vector1*matrixv
> # Or
> matrixv*vector1
> # Multiply rows of matrix by vector
> t(vector1*t(matrixv))
> # Multiply rows of matrix by vector - transpose is very slow
> product <- lapply(1:NCOL(matrixv),
+   function(x) vector1[x]*matrixv[, x])
> do.call(cbind, product)
> library(microbenchmark)
> summary(microbenchmark(
+   trans=t(vector1*t(matrixv)),
+   lapp={
+     product <- lapply(1:NCOL(matrixv), function(x) vector1[x]*mat
+     do.call(cbind, product)
+   },
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Matrix Inner Multiplication

The `%*%` operator performs *inner* (*scalar*) multiplication of vectors and matrices.

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so that each pair produces a single number:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$$

Inner multiplication produces a vector or matrix with a reduced dimension.

Inner multiplication requires the dimensions of the matrices to be *conformable* (number of columns in the first matrix must be equal to the number of rows in the second).

The function `drop()` removes any dimensions of length *one*.

The functions `rowSums()` and `colSums()` calculate the sums of rows and columns, and they're very fast because they pass their data to compiled C++ code.

```
> vector2 <- 6:4  # Define vector
> # Multiply two vectors element-by-element
> vector1 * vector2
> # Calculate inner product
> vector1 %*% vector2
> # Calculate inner product and drop dimensions
> drop(vector1 %*% vector2)
> # Multiply columns of matrix by vector
> matrixv %*% vector1  # Single column matrix
> drop(matrixv %*% vector1)  # vector
> rowSums(t(vector1 * t(matrixv)))
> # using rowSums() and t() is 10 times slower than %*%
> library(microbenchmark)
> summary(microbenchmark(
+    inner=drop(matrixv %*% vector1),
+    transp=rowSums(t(vector1 * t(matrixv))),
+    times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Matrix Transpose

The function t() returns the transpose of a matrix.

The function crossprod() also performs *inner* (*scalar*) multiplication, exactly the same as the %*% operator, but is slightly faster.

```
> # Multiply matrix by vector fails because dimensions aren't confor
> vector1 %*% matrixv
> # Works after transpose
> drop(vector1 %*% t(matrixv))
> # Calculate inner product
> crossprod(vector1, vector2)
> # Create matrix and vector
> matrixv <- matrix(1:3000, ncol=3)
> tmatrixv <- t(matrixv)
> vectorv <- 1:3
> # crossprod() is slightly faster than "%*%" operator
> summary(microbenchmark(
+   cross_prod=crossprod(tmatrixv, vectorv),
+   inner_prod=matrixv %*% vectorv,
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Matrix Outer Multiplication

An *outer* product consists of all possible products of pairs of elements of two objects:

$$C_{i,j} = A_i \cdot B_j$$

An *outer* product of a function consists of applying it to all possible pairs of elements of two objects:

$$C_{i,j} = f(A_i, B_j)$$

Outer multiplication produces an object with dimension equal to the sum of the factors' dimensions, and with the number of elements equal to the product of the factors' elements.

The function `outer()` calculates the *outer* product of two matrices, and by default multiplies the elements of its arguments.

`outer()` can also calculate the values of a vectorized function of two variables passed to the "FUN" argument.

```
> # Define named vectors
> vector1 <- sample(1:4)
> names(vector1) <- paste0("row", 1:4, "=", vector1)
> vector1
> vector2 <- sample(1:3)
> names(vector2) <- paste0("col", 1:3, "=", vector2)
> vector2
> # Calculate outer product of two vectors
> matrixv <- outer(vector1, vector2)
> matrixv
> # Calculate vectorized function spanned over two vectors
> matrixv <- outer(vector1, vector2,
+         FUN=function(x1, x2) x2*sin(x1))
> matrixv
```

Functions in R

R functions have three components:

- a list of formal arguments,
- a body containing R code,
- an environment,

An R function plus its environment is referred to as a function *closures*.

The function body should be enclosed in curly braces {}, unless it contains a single command, then it doesn't have to enclosed.

The function body doesn't require a return statement, since by default R functions return the last statement evaluated in the body.

args() displays the formal arguments of a function.

```
> # Define a function with two arguments
> testfun <- function(first_arg, second_arg) {  # Body
+   first_arg + second_arg # Returns last evaluated statement
+ }  # end testfun
>
> testfun(1, 2)  # Apply the function
> args(testfun)  # Display argument
>
> # Define function that uses variable from enclosure environment
> testfun <- function(first_arg, second_arg) {
+   first_arg + second_arg + globv
+ }  # end testfun
>
> testfun(3, 2)  # error - globv doesn't exist yet!
> globv <- 10  # Create globv
> testfun(3, 2)  # Now works
```

Return Values of Functions

The function body doesn't require a `return` statement, since by default R functions return the last statement evaluated in the body.

`return()` statements are inserted in logical branches to terminate function execution and return its intended value.

```
> # Define function that returns NULL for non-numeric argument
> testfun <- function(input) {
+   if (!is.numeric(input)) {
+     warning(paste("argument", input, "isn't numeric"))
+     return(NULL)
+   }
+   2*input
+ }  # end testfun
>
> testfun(2)
> testfun("hello")
```

Functions That Return `invisible`

If a return value is wrapped in the function `invisible()` then the return value isn't printed.

But if the function is assigned to a variable, then its return value is assigned to that variable.

`invisible()` allows creating functions whose return values can be assigned, but which do not print when they're not assigned.

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

```
> # Define a function that returns invisibly
> return_invisible <- function(input) {
+   invisible(input)
+ }  # end return_invisible
>
> return_invisible(2)
>
> globv <- return_invisible(2)
> globv
>
> rm(list=ls())  # Remove all objects
> # Load objects from file
> loaded <- load(file="/Users/jerzy/Develop/data/my_data.RData")
> loaded  # Vector of loaded objects
> ls()  # List objects
```

Binding Function Arguments

The formal arguments of a function are defined in its argument list.

When a function is called, it's passed a list of actual function arguments.

Formal arguments can be *bound* to actual arguments either by name or by position:

- by name: formal arguments are *bound* to actual arguments with the same name,

- by position: the first formal argument is *bound* to the first actual argument, etc.

Binding by name takes precedence over *binding* by position: first all the named arguments are *bound*, then the remaining arguments are *bound* by position.

Partial argument names are *bound* to full names.

```
> testfun <- function(first_arg, second_arg) {
+ # Last statement of function is return value
+   first_arg + 2*second_arg
+ }  # end testfun
> testfun(first_arg=3, second_arg=2)  # Bind by name
> testfun(first=3, second=2)  # Partial name binding
> testfun(3, 2)  # Bind by position
> testfun(second_arg=2, 3)  # mixed binding
> testfun(3, 2, 1)  # Too many arguments
> testfun(2)  # Not enough arguments
```

All the actual arguments must be *bound* to formal arguments, and if not then an "unused argument" error is produced.

If there aren't enough formal arguments, then an "argument is missing" error is produced,

Default Values for Arguments

Formal arguments may be assigned default values, so that when the actual arguments are missing then their default values are used instead.

Default values are often assigned to function parameters, that determine the function's behavior.

Default values can be specified as a vector of strings, representing the possible values of a function's parameter.

The function `match.arg()` matches a string to one of the possible values, and returns the matched value, or produces an `error` if it can't match it.

The function `str()` displays the structure of an R object, for example a function name and its formal arguments.

```
> # Function "paste" has two arguments with default values
> str(paste)
> # Default values of arguments can be specified in argument list
> testfun <- function(first_arg, ratio=1) {
+    ratio*first_arg
+ }  # end testfun
> testfun(3)  # Default value used for second argument
> testfun(3, 2)  # Default value over-ridden
> # Default values can be a vector of strings
> testfun <- function(input=c("first_val", "second_val")) {
+    input <- match.arg(input)  # Match to arg list
+    input
+ }  # end testfun
> testfun("second_val")
> testfun("se")  # Partial name binding
> testfun("some_val")  # Invalid string
```

Function for Calculating Skew

R provides an easy way for users to write functions.

Formal function arguments can be bound to input variables by position or by name.

If the function arguments are missing then their default value is used.

Functions return the value of the last expression that is evaluated.

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

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

```
> # DAX percentage returns
> returns <- rutils::diffit(log(EuStockMarkets[, 1]))
> # calc_skew() calculates skew of time series of returns
> # Default is normal time series
> calc_skew <- function(returns=rnorm(1000)) {
+   # Number of observations
+   nrows <- NROW(returns)
+   # Standardize returns
+   returns <- (returns - mean(returns))/sd(returns)
+   # Calculate skew - last statement automatically returned
+   .n_rows*sum(returns^3)/((nrows-1)*(nrows-2))
+ }  # end calc_skew
>
> # Calculate skew of DAX returns
> # Bind arguments by name
> calc_skew(returns=returns)
> # Bind arguments by position
> calc_skew(returns)
> # Use default value of arguments
> calc_skew()
```

The dots "..." Function Argument

The dots "..." function argument is a formal argument without a name, as opposed to the other formal arguments which all have names.

The dots "..." bind with any number of additional arguments, that aren't already bound by name or position to the named arguments.

The dots "..." are used when the number of arguments isn't known in advance, and allows functions to accept an indefinite number of arguments.

The dots "..." are sometimes placed *after* the named arguments, to allow passing of additional parameters into a function.

Functionals often place the dots "..." argument *after* the named arguments, to allow passing the dots "..." to the function being called by the *functional*.

```
> str(plot)  # Dots for additional plot parameters
> bind_dots <- function(input, ...) {
+   paste0("input=", input,
+   ", dots=", paste(..., sep=", "))
+ }  # end bind_dots
> bind_dots(1, 2, 3)  # "input" bound by position
> bind_dots(2, input=1, 3)  # "input" bound by name
> bind_dots(1, 2, 3, foo=10)  # Named argument bound to dots
> bind_dots <- function(arg1, arg2, ...) {
+   arg1 + 2*arg2 + sum(...)
+ }  # end bind_dots
> bind_dots(3, 2)  # Bind arguments by position
> bind_dots(3, 2, 5, 8)  # Extra arguments bound to dots
```

Argument Binding With dots "..." Argument

The dots "..." argument is sometimes placed *before* the named arguments, so that a function can accept an indefinite number of arguments, without binding them by position with the named arguments.

When the dots "..." are placed *before* the named arguments, the named arguments are often assigned default values, so they don't have to be bound to a value in the call.

Arguments that appear after the dots "..." must be *bound* by their full name, and can't be partially *bound*.

```
> str(sum)  # Dots before other arguments
> sum(1, 2, 3)  # Dots bind before other arguments
> sum(1, 2, NA, 3, na.rm=TRUE)
> bind_dots <- function(..., input) {
+   paste0("input=", input,
+   ", dots=", paste(..., sep=", "))
+ }  # end bind_dots
> # Arguments after dots must be bound by full name
> bind_dots(1, 2, 3, input=10)
> bind_dots(1, 2, 3, input=10, foo=4)  # Dots bound
> bind_dots(1, 2, 3)  # "input" not bound
> bind_dots <- function(..., input=10) {
+   paste0("input=", input,
+   ", dots=", paste(..., sep=", "))
+ }  # end bind_dots
> bind_dots(1, 2, 3)  # "input" not bound, but has default
```

Wrapper Functions With dots "..." Argument

Wrapper functions provide a convenient user interface to functions, by assigning default argument values, validating data, and formatting the output.

Wrapper functions are designed to perform the actions of other functions, while reducing their complexity.

The dots "..." argument of the *wrapper* function allows passing additional arguments on to the wrapped function.

Wrapper functions should be used with caution, since wrapping a function creates extra code (overhead), which slows down R.

```
> # Wrapper for mean() with default na.rm=TRUE
> my_mean <- function(x, na.rm=TRUE, ...) {
+   mean(x=x, na.rm=na.rm, ...)
+ }  # end my_mean
> foo <- sample(c(1:10, NA, rep(0.1, t=5)))
> mean(c(foo, NA))
> mean(c(foo, NA), na.rm=TRUE)
> my_mean(c(foo, NA))
> my_mean(c(foo, NA), trim=0.4)  # Pass extra argument
> # Wrapper for saving data into default directory
> save_data <- function(...,
+                 file=stop("error: no file name"),
+                 my_dir="/Users/jerzy/Develop/data") {
+ # Create file path
+   file <- file.path(my_dir, file)
+   save(..., file=file)
+ }  # end save_data
> foo <- 1:10
> save_data(foo, file="scratch.RData")
> save_data(foo, file="scratch.RData", my_dir="/Users/jerzy/Develop'
> # Wrapper for testing negative arguments
> stop_if_neg <- function(input) {
+   if (!is.numeric(input) || input<0)
+     stop("argument not numeric or negative")
+ }  # end stop_if_neg
> # Wrapper for sqrt()
> my_sqrt <- function(input) {
+   stop_if_neg(input)
+   sqrt(input)
+ }  # end my_sqrt
> my_sqrt(2)
> my_sqrt(-2)
> my_sqrt(NA)
```

Recursive Functions with dots "..." Argument

Recursive functions can also accept the dots "..." argument.

The dots "..." argument can be referenced inside a function by first converting it into a list using "list(...)".

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

```
> # Recursive function sums its argument list
> sum_dots <- function(input, ...) {
+   if (missing(...)) {  # Check if dots are empty
+     return(input)  # just one argument left
+   } else {
+     input + sum_dots(...)  # Sum remaining arguments
+   }  # end if
+ }  # end sum_dots
> sum_dots(1, 2, 3, 4)
> # Recursive function sums its argument list
> sum_dots <- function(input, ...) {
+   if (NROW(list(...)) == 0) {  # Check if dots are empty
+     return(input)  # just one argument left
+   } else {
+     input + sum_dots(...)  # Sum remaining arguments
+   }  # end if
+ }  # end sum_dots
> sum_dots(1, 2, 3, 4)
```

Recursive Function for Calculating Fibonacci Sequence

Recursive functions call themselves in their own body.

The *Fibonacci* sequence of integers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2},$$
$$F1 = 0, F2 = 1,$$
$$F_n = 0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

The *Fibonacci* sequence was invented by *Indian* mathematicians, and later described by the Italian mathematician *Fibonacci* in his famous treatise *Liber Abaci*.

```
> fibonacci <- function(nrows) {
+    if (nrows > 2) {
+       fib_seq <- fibonacci(nrows-1)  # Recursion
+       c(fib_seq, sum(tail(fib_seq, 2)))  # Return this
+    } else {
+       c(0, 1)  # Initialize and return
+    }
+ }  # end fibonacci
> fibonacci(10)
> tail(fibonacci(9), 2)
```

Exploring Functions

If a function name is called alone without arguments, then R displays the function code (but it must be on the search path).

Non-visible objects can't be viewed by calling their name.

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

The function getAnywhere() also displays R objects that aren't on the search path.

```
> # Show the function code
> plot.default
> # Display function
> getAnywhere(plot.default)
```

Function Environments

When a function is called, a new *evaluation* environment is created.

The *evaluation* environment contains the function arguments and locally defined variables.

R evaluates variables inside functions by searching first in the *evaluation* environment, then the *enclosure* environment, then the R search path.

The enclosure of the *evaluation* environment is the environment where the function was defined.

The enclosure of functions defined in the workspace is the *global* environment.

The enclosure of functions defined in packages is the package *namesvpace*.

Objects defined in the function enclosure can be referenced inside the function.

```
> globv <- 1  # Define a global variable
> ls(environment())  # Get all variables in environment
> func_env <- function() {  # Explore function environments
+   locvar <- 1  # Define a local variable
+   cat('objects in evaluation environment:\t',
+       ls(environment()), '\n')
+   cat('objects in enclosing environment:\t',
+       ls(parent.env(environment())), '\n')
+   cat('this is the enclosing environment:')
+   parent.env(environment())  # Return enclosing environment
+ }  # end func_env
> func_env()
>
> environment(func_env)
> environment(print)  # Package namespace is the enclosure
```

Side effects Using the Super-assignment Operator "<<-"

Function *side effects* are operations on objects outside a function's *evaluation* environment.

The functions `plot()` and `load()` are examples of functions that produce *side effects*.

`load()` reads data from an `.RData` file, and creates objects in the workspace that are contained in the `.RData` file.

The super-assignment operator "<<-" allows creating functions that produce *side effects*.

The super-assignment operator "<<-" modifies or creates variables in the *enclosing* environment in which a function was *defined* (*lexical* scoping).

If a function was *defined* in the *global* environment then that's the function's *enclosing* environment, and the "<<-" operator operates on variables in the *global* environment.

```
> rm(list=ls())  # Remove all objects
> ls()  # List objects
> # Load objects from file (side effect)
> load(file="my_data.RData")
> ls()  # List objects
> globv <- 1  # Define a global variable
> # Explore function scope and side effects
> side_effect <- function() {
+   cat("global globv =", globv, "\n")
+ # Define local "globv" variable
+   globv <- 10
+   cat("local globv =", globv, "\n")
+   # Re-define the global "globv"
+   globv <<- 2
+   cat("local globv =", globv, "\n")
+ }  # end side_effect
> side_effect()
> # Global variable was modified as side effect
> globv
```

Functions as First Class Objects

Functions in R are *first class objects*, which means they can be treated like any other R object:

- Functions can be passed as arguments to other functions,
- Functions can be nested (defined inside other functions),
- Functions can return functions as their return value,

Higher order functions are R functions that either accept a function as their argument (input) or return a function as their value (output).

```
> # Create functional that accepts a function as input argument
> testfun <- function(func_name) {
+ # Calculates statistic on random numbers
+   set.seed(1)
+   func_name(runif(1e4))  # Apply the function name
+ }  # end testfun
> testfun(mean)
> testfun(sd)
```

Functionals

Functionals are functions that accept a function or a function name (string) as one of their input arguments.

Functionals are able to execute function calls using the function names.

The function `match.fun()` returns a function name that is specified by a string.

Functionals that call `match.fun()` are able to accept a string as a function name, because `match.fun()` converts it to a function.

`match.fun()` produces an error condition if it fails to find a function with the specified name.

```
> # Functional accepts function name and additional argument
> testfun <- function(func_name, input) {
+ # Produce function name from argument
+   func_name <- match.fun(func_name)
+ # Execute function call
+   func_name(input)
+ }  # end testfun
> testfun(sqrt, 4)
> # String also works because match.fun() converts it to a function
> testfun("sqrt", 4)
> str(sum)  # Sum() accepts multiple arguments
> # Functional can't accept indefinite number of arguments
> testfun(sum, 1, 2, 3)
```

Functionals with dots "..." Argument

The dots "..." argument in *functionals* can be used to pass additional arguments to the function being called by the *functional*.

If named values are passed to the dots "..." argument, then the *functional* can bind them to the correct formal arguments of the function being called by the *functional*.

```
> # Functional accepts function name and dots '...' argument
> testfun <- function(func_name, ...) {
+   func_name <- match.fun(func_name)
+   func_name(...)  # Execute function call
+ }  # end testfun
> testfun(sum, 1, 2, 3)
> testfun(sum, 1, 2, NA, 4, 5)
> testfun(sum, 1, 2, NA, 4, 5, na.rm=TRUE)
> # Function with three arguments and dots '...' arguments
> testfun <- function(input, param1, param2, ...) {
+   c(input=input, param1=param1, param2=param2, dots=c(...))
+ }  # end testfun
> testfun(1, 2, 3, param2=4, param1=5)
> testfun(testfun, 1, 2, 3, param2=4, param1=5)
> testfun(testfun, 1, 2, 3, 4, 5)
```

Anonymous Functions

R allows defining functions without assigning a name to them.

Anonymous functions are functions that are not assigned to a name.

Anonymous functions can be passed as arguments to *functionals*.

```
> # Simple anonymous function
> (function(x) (x + 3)) (10)
```

Functionals with Anonymous Functions

Anonymous functions can be passed as arguments to *functionals*.

Anonymous functions can also be used as default values for function arguments.

```
> # Anonymous function passed to testfun
> testfun(func_name=(function(x) (x + 3)), 5)
> # Anonymous function is default value
> testfun <-
+   function(..., func_name=function(x, y, z) {x+y+z}) {
+     func_name <- match.fun(func_name)
+     func_name(...)  # Execute function call
+ }  # end testfun
> testfun(2, 3, 4)  # Use default func_name
> testfun(2, 3, 4, 5)
> # Func_name bound by name
> testfun(func_name=sum, 2, 3, 4, 5)
> # Pass anonymous function to func_name
> testfun(func_name=function(x, y, z) {x*y*z},
+     2, 3, 4)
```

Executing Function Calls Using the `do.call()` Functional

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

do.call() allows calling a function on arguments that are elements of a list.

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

do.call() can be called inside other *functionals* to allow them to execute function calls.

The function str() displays the structure of an R object, for example a function name and its formal 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.

```
> str(sum)  # Sum() accepts multiple arguments
> # Sum() can't accept list of arguments
> sum(list(1, 2, 3))
> str(do.call)  # "what" argument is a function
> # Do.call passes list elements into "sum" individually
> do.call(sum, list(1, 2, 3))
> do.call(sum, list(1, 2, NA, 3))
> do.call(sum, list(1, 2, NA, 3, na.rm=TRUE))
> # Functional accepts list with function name and arguments
> testfun <- function(list_arg) {
+ # Produce function name from argument
+   func_name <- match.fun(list_arg[[1]])
+ # Execute function call uing do.call()
+   do.call(func_name, list_arg[-1])
+ }  # end testfun
> arg_list <- list("sum", 1, 2, 3)
> testfun(arg_list)
> # do_call() performs same operation as do.call()
> all.equal(
+   do.call(sum, list(1, 2, NA, 3, na.rm=TRUE)),
+   rutils::do_call(sum, list(1, 2, NA, 3), na.rm=TRUE))
```

Performing Loops Using the `apply()` Functionals

An important example of *functionals* are the `apply()` functionals.

The functional `apply()` returns the result of applying a function to the rows or columns of an array or matrix.

If `MARGIN=1` then the function will be applied over the matrix *rows,*

If `MARGIN=2` then the function will be applied over the matrix *columns.*

`apply()` performs a loop over the list of objects, and can replace `"for"` loops in R.

```
> str(apply)  # Get list of arguments
> # Create a matrix
> matrixv <- matrix(6:1, nrow=2, ncol=3)
> matrixv
> # Sum the rows and columns
> rowsumv <- apply(matrixv, 1, sum)
> colsumv <- apply(matrixv, 2, sum)
> matrixv <- cbind(c(sum(rowsumv), rowsumv),
+         rbind(colsumv, matrixv))
> dimnames(matrixv) <- list(c("colsumv", "row1", "row2"),
+              c("rowsumv", "col1", "col2", "col3"))
> matrixv
```

The apply() Functional with dots "..." Argument

The dots "..." argument in apply() is designed to pass additional arguments to the function being called by apply().

The additional arguments to apply() must be *bound* by their full (complete) names.

```
> str(apply)  # Get list of arguments
> matrixv <- matrix(sample(12), nrow=3, ncol=4)  # Create a matrix
> matrixv
> apply(matrixv, 2, sort)  # Sort matrix columns
> apply(matrixv, 2, sort, decreasing=TRUE)  # Sort decreasing order
```

```
> matrixv[2, 2] <- NA  # Introduce NA value
> matrixv
> # Calculate median of columns
> apply(matrixv, 2, median)
> # Calculate median of columns with na.rm=TRUE
> apply(matrixv, 2, median, na.rm=TRUE)
```

The apply() Functional with Anonymous Functions

The apply() functional combined with *anonymous* functions can be used to loop over function parameters.

The dots "..." argument in apply() is designed to pass additional arguments to the function being called by apply().

The additional arguments to apply() must be *bound* by their full (complete) names.

```
> # DAX percentage returns
> returns <- rutils::diffit(log(EuStockMarkets[, 1]))
> library(moments)  # Load package moments
> str(moment)  # Get list of arguments
> # Apply moment function
> moment(x=returns, order=3)
> # 4x1 matrix of moment orders
> morderv <- as.matrix(1:4)
> # Anonymous function allows looping over function parameters
> apply(X=morderv, MARGIN=1,
+       FUN=function(moment_order) {
+   moment(x=returns, order=moment_order)
+ }  # end anonymous function
+       )  # end apply
>
> # Another way of passing parameters into moment() function
> apply(X=morderv, MARGIN=1, FUN=moment, x=returns)
```

apply() Calling Functions with Multiple Arguments

When apply() calls a function with multiple arguments, then care must be taken for proper argument binding.

The dots "..." argument in apply() allows passing additional arguments to the function being called by apply().

The additional arguments to apply() must be *bound* by their full (complete) names.

The values of the "X" argument in apply() are *bound* by position to the first unused argument in the function being called by apply().

```
> # Function with three arguments
> testfun <- function(arg1, arg2, arg3) {
+   c(arg1=arg1, arg2=arg2, arg3=arg3)
+ }  # end testfun
> testfun(1, 2, 3)
> datav <- as.matrix(1:4)
> # Pass datav to arg1
> apply(X=datav, MAR=1, FUN=testfun, arg2=2, arg3=3)
> # Pass datav to arg2
> apply(X=datav, MAR=1, FUN=testfun, arg1=1, arg3=3)
> # Pass datav to arg3
> apply(X=datav, MAR=1, FUN=testfun, arg1=1, arg2=2)
```

The `lapply()` Functional

The functional `lapply()` is a specialized version of the functional `apply()`.

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

The function `unlist()` collapses a list with atomic elements into a vector (which can cause type coercion).

Rule of Thumb
It's often better to use `lapply()`, since `apply()` and `sapply()` attempt to coerce their output into a vector or matrix, which may cause them to fail.

```
> # Vector of means of numeric columns
> sapply(iris[, -5], mean)
> # List of means of numeric columns
> lapply(iris[, -5], mean)
> # Lapply using anonymous function
> unlist(lapply(iris,
+        function(colnum) {
+          if (is.numeric(colnum)) mean(colnum)
+        }  # end anonymous function
+      )  # end lapply
+    )  # end unlist
> unlist(sapply(iris, function(colnum) {
+    if (is.numeric(colnum)) mean(colnum)}))
```

The sapply() Functional

The sapply() functional is a specialized version of the apply() functional.

sapply() applies a function to a vector or a list of objects and returns a vector or a list.

sapply() tries to return a vector, but if the elements can't be combined into a vector, then it returns a list.

When sapply() is given a data frame, it interretsp it as a list, and applies the function to each element (column) of the data frame.

```
> sapply(6:10, sqrt)  # Sapply on vector
> sapply(list(6, 7, 8, 9, 10), sqrt)  # Sapply on list
>
> # Calculate means of iris data frame columns
> sapply(iris, mean)  # Returns NA for Species
>
> # Create a matrix
> matrixv <- matrix(sample(100), ncol=4)
> # Calculate column means using apply
> apply(matrixv, 2, mean)
>
> # Calculate column means using sapply, with anonymous function
> sapply(1:NCOL(matrixv),
+        function(col_index) {  # Anonymous function
+   mean(matrixv[, col_index])
+   }  # end anonymous function
+ )  # end sapply
```

sapply() Returning Matrices

If the function called by `sapply()` returns a vector, then `sapply()` returns a matrix, if possible.

The vectors returned by the function are arranged to form columns of the matrix returned by `sapply()`.

But if the function returns vectors of different lengths, then `sapply()` cannot return a matrix, and returns a list instead.

This behavior of `sapply()` can cause run-time errors.

The function `vapply()` is similar to `sapply()`, but it always attempts to simplify its output to a matrix, and if it can't then it produces an error.

`vapply()` requires the argument `FUN.VALUE` that specifes the output format of the function called by `vapply()`.

```
> # Vectors form columns of matrix returned by sapply
> sapply(2:4, function(num) c(el1=num, el2=2*num))
> # Vectors of different lengths returned as list
> sapply(2:4, function(num) 1:num)
> # vapply is similar to sapply
> vapply(2:4, function(num) c(el1=num, el2=2*num),
+        FUN.VALUE=c(row1=0, row2=0))
> # vapply produces an error if it can't simplify
> vapply(2:4, function(num) 1:num,
+        FUN.VALUE=c(row1=0, row2=0))
```

Normal (Gaussian) Probability Distribution

The *Normal (Gaussian)* probability density function is given by:

$$\phi(x, \mu, \sigma) = \frac{e^{-(x-\mu)^2/2\sigma^2}}{\sigma\sqrt{2\pi}}$$

The *Standard Normal* distribution $\phi(0, 1)$ is a special case of the *Normal* $\phi(\mu, \sigma)$ with $\mu = 0$ and $\sigma = 1$.

The function `dnorm()` calculates the *Normal* probability density.

```
> xvar <- seq(-5, 7, length=100)
> yvar <- dnorm(xvar, mean=1.0, sd=2.0)
> plot(xvar, yvar, type="l", lty="solid",
+      xlab="", ylab="")
> title(main="Normal Density Function", line=0.5)
> startp <- 3; endp <- 5  # Set lower and upper bounds
> # Set polygon base
> subv <- ((xvar >= startp) & (xvar <= endp))
> polygon(c(startp, xvar[subv], endp),  # Draw polygon
+    c(-1, yvar[subv], -1), col="red")
```


Normal Density Function
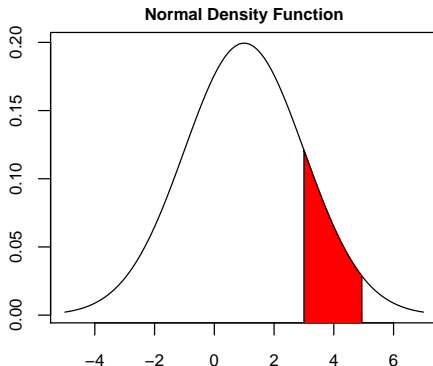
Normal (Gaussian) Probability Distributions

Plots of several *Normal* distributions with different values of σ, using the function `curve()` for plotting functions given by their name.

```
> sigmavs <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> colors <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("sigma", sigmavs, sep="=")
> for (indeks in 1:4) {  # Plot four curves
+ curve(expr=dnorm(x, sd=sigmavs[indeks]),
+ xlim=c(-4, 4),
+ xlab="", ylab="", lwd=2,
+ col=colors[indeks],
+ add=as.logical(indeks-1))
+ }  # end for
> # Add title
> title(main="Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, title="Sigmas",
+ labelv, cex=0.8, lwd=2, lty=1, bty="n",
+ col=colors)
```
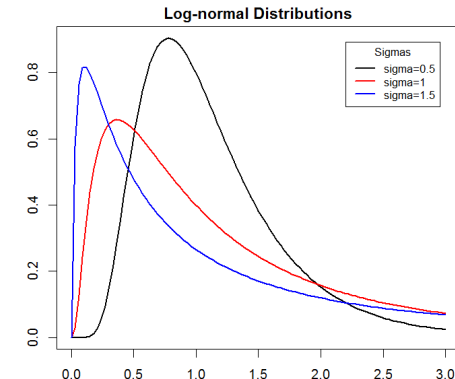

Normal Distributions
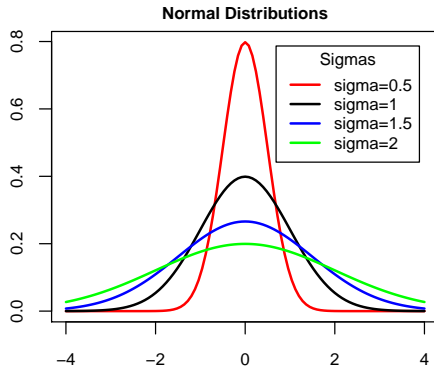
Normal Probability Distributions Plotted as Lines

Plots of several *Normal* distributions with different values of σ.

```
> xvar <- seq(-4, 4, length=100)
> sigmavs <- c(0.5, 1, 1.5, 2)  # Sigma values
> # Create plot colors
> colors <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("sigma", sigmavs, sep="=")
> # Plot the first chart
> plot(xvar, dnorm(xvar, sd=sigmavs[1]),
+      type="n", xlab="", ylab="",
+      main="Normal Distributions")
> # Add lines to plot
> for (indeks in 1:4) {
+   lines(xvar, dnorm(xvar, sd=sigmavs[indeks]),
+   lwd=2, col=colors[indeks])
+ }  # end for
> # Add legend
> legend("topright", inset=0.05, title="Sigmas",
+   labelv, cex=0.8, lwd=2, lty=1, bty="n",
+   col=colors)
```
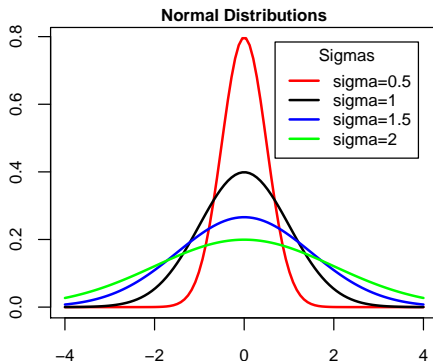
The *Log-normal* Probability Distribution

If x follows the *Normal* distribution $\phi(x, \mu, \sigma)$, then the exponential of x: $y = e^x$ follows the *Log-normal* distribution $\log \phi()$:

$$\log \phi(y, \mu, \sigma) = \frac{\exp(-(\log y - \mu)^2 / 2\sigma^2)}{y \sigma \sqrt{2\pi}}$$

With mean equal to: $\bar{y} = \mathbb{E}[y] = \exp(\mu + \sigma^2/2)$, and median equal to: $\tilde{y} = \exp(\mu)$

```
> # Standard deviations of log-normal distribution
> sigmavs <- c(0.5, 1, 1.5)
> # Create plot colors
> colors <- c("black", "red", "blue")
> # Plot all curves
> for (indeks in 1:NROW(sigmavs)) {
+   curve(expr=dlnorm(x, sdlog=sigmavs[indeks]),
+   type="l", xlim=c(0, 3), lwd=2,
+   xlab="", ylab="", col=colors[indeks],
+   add=as.logical(indeks-1))
+ }  # end for
```


Log-normal Distributions

```
> # Add title and legend
> title(main="Log-normal Distributions", line=0.5)
> legend("topright", inset=0.05, title="Sigmas",
+   paste("sigma", sigmavs, sep="="),
+   cex=0.8, lwd=2, lty=rep(1, NROW(sigmavs)),
+   col=colors)
```
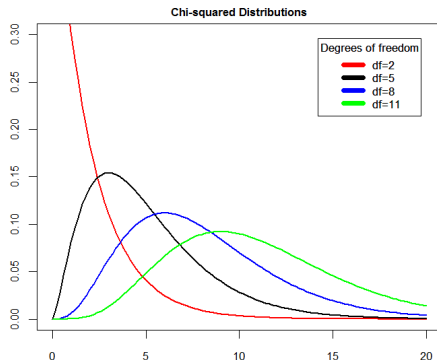
Chi-squared Distribution

Let $z1, \ldots, z_k$ be independent standard *Normal* random variables.

Then the random variable $X = \sum_{i=1}^{k} z_i^2$ is distributed according to the *Chi-squared* distribution with k degrees of freedom: $X \sim \chi_k^2$, and its probability density function is given by:

$$f(x) = \frac{x^{k/2-1} e^{-x/2}}{2^{k/2} \, \Gamma(k/2)}$$

The *Chi-squared* distribution with k degrees of freedom has mean equal to k and variance equal to $2k$.

```
> # Degrees of freedom
> degf <- c(2, 5, 8, 11)
> # Plot four curves in loop
> colors <- c("red", "black", "blue", "green")
> for (indeks in 1:4) {
+ curve(expr=dchisq(x, df=degf[indeks]),
+ xlim=c(0, 20), ylim=c(0, 0.3),
+ xlab="", ylab="", col=colors[indeks],
+ lwd=2, add=as.logical(indeks-1))
+ }  # end for
```


Chi-squared Distributions

```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> labelv <- paste("df", degf, sep="=")
> legend("topright", inset=0.05, bty="n",
+        title="Degrees of freedom", labelv,
+        cex=0.8, lwd=6, lty=1,
+        col=colors)
```
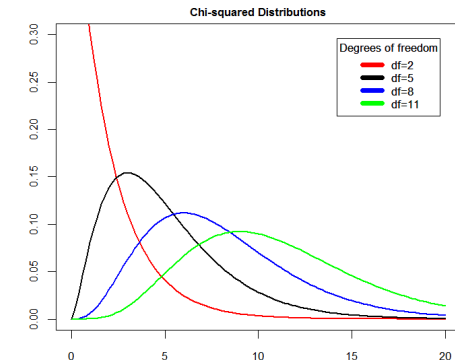
Chi-squared Distribution Plotted as Line

Let $z_1, \ldots, z_k$ be independent standard *Normal* random variables.

Then the random variable $X = \sum_{i=1}^{k} z_i^2$ is distributed according to the *Chi-squared* distribution with k degrees of freedom: $X \sim \chi_k^2$, and its probability density function is given by:

$$f(x) = \frac{x^{k/2-1} e^{-x/2}}{2^{k/2} \Gamma(k/2)}$$

Chi-squared Distributions

```
> degf <- c(2, 5, 8, 11)  # df values
> # Create plot colors
> colors <- c("red", "black", "blue", "green")
> # Create legend labels
> labelv <- paste("df", degf, sep="=")
> # Plot an empty chart
> xvar <- seq(0, 20, length=100)
> plot(xvar, dchisq(xvar, df=degf[1]),
+      type="n", xlab="", ylab="", ylim=c(0, 0.3))
> # Add lines to plot
> for (indeks in 1:4) {
+   lines(xvar, dchisq(xvar, df=degf[indeks]),
+ lwd=2, col=colors[indeks])
+ }  # end for
```

```
> # Add title
> title(main="Chi-squared Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05,
+        title="Degrees of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, bty="n", col=colors)
```

Fisher's *F-distribution*

Let χ_m^2 and χ_n^2 be independent random variables following *chi-squared* distributions with m and n degrees of freedom.

Then the random variable:

$$F = \frac{\chi_m^2/m}{\chi_n^2/n}$$

Follows the *F-distribution* with m and n degrees of freedom, with the probability density function:

$$f(F) = \frac{\Gamma((m+n)/2)m^{m/2}n^{n/2}}{\Gamma(m/2)\Gamma(n/2)} \frac{F^{m/2-1}}{(n+mF)^{(m+n)/2}}$$

The *F-distribution* depends on the ratio F and also on the degrees of freedom, m and n.

The function df() calculates the probability density of the *F-distribution*.

```
> # Plot three curves in loop
> degf <- c(3, 5, 9)  # Degrees of freedom
> colors <- c("black", "red", "blue", "green")
> for (indeks in 1:NROW(degf)) {
+ curve(expr=df(x, df1=degf[indeks], df2=3),
+ xlim=c(0, 4), xlab="", ylab="", lwd=2,
+ col=colors[indeks], add=as.logical(indeks-1))
+ }  # end for
```
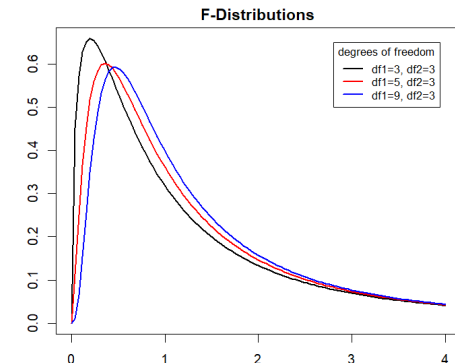

F-Distributions

```
> # Add title
> title(main="F-Distributions", line=0.5)
> # Add legend
> labelv <- paste("df", degf, sep="=")
> legend("topright", inset=0.05, title="degrees of freedom",
+        labelv, cex=0.8, lwd=2, lty=1,
+        col=colors)
```

Student's *t-distribution*

Let $z1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu} \sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance: $\hat{\sigma}^2 = \frac{1}{\nu-1} \sum_{i=1}^{\nu} (z_i - \bar{z})^2$
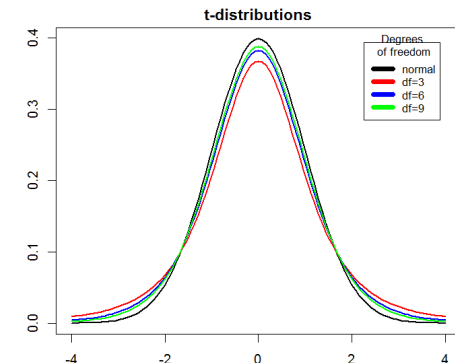
Then the random variable (*t-ratio*):

$$t = \frac{\bar{z} - \mu}{\hat{\sigma}/\sqrt{\nu}}$$

Follows the *t-distribution* with ν degrees of freedom, with the probability density function:

$$f(t) = \frac{\Gamma((\nu + 1)/2)}{\sqrt{\pi \nu}\, \Gamma(\nu/2)} \left(1 + t^2/\nu\right)^{-(\nu+1)/2}$$

```
> degf <- c(3, 6, 9)  # df values
> colors <- c("black", "red", "blue", "green")
> labelv <- c("normal", paste("df", degf, sep="="))
> # Plot a Normal probability distribution
> curve(expr=dnorm, xlim=c(-4, 4),
+       xlab="", ylab="", lwd=2)
> for (indeks in 1:3) {  # Plot three t-distributions
+ curve(expr=dt(x, df=degf[indeks]),
+       lwd=2, col=colors[indeks+1], add=TRUE)
+ }  # end for
```



```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title="Degrees\n of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colors)
```

Student's *t-distribution* Plotted as Line

Let $z1, \ldots, z_\nu$ be independent standard normal random variables, with sample mean: $\bar{z} = \frac{1}{\nu} \sum_{i=1}^{\nu} z_i$ ($\mathbb{E}[\bar{z}] = \mu$) and sample variance:
$\hat{\sigma}^2 = \frac{1}{\nu-1} \sum_{i=1}^{\nu} (z_i - \bar{z})^2$

Then the random variable (*t-ratio*):

$$t = \frac{\bar{z} - \mu}{\hat{\sigma}/\sqrt{\nu}}$$

Follows the *t-distribution* with ν degrees of freedom, with the probability density function:

$$f(t) = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)} \left(1 + t^2/\nu\right)^{-(\nu+1)/2}$$

t-distributions

```
> xvar <- seq(-4, 4, length=100)
> degf <- c(3, 6, 9)  # df values
> colors <- c("black", "red", "blue", "green")
> labelv <- c("normal", paste("df", degf, sep="="))
> # Plot chart of normal distribution
> plot(xvar, dnorm(xvar), type="l",
+      lwd=2, xlab="", ylab="")
> for (indeks in 1:3) {  # Add lines for t-distributions
+   lines(xvar, dt(xvar, df=degf[indeks]),
+ lwd=2, col=colors[indeks+1])
+ }  # end for
```
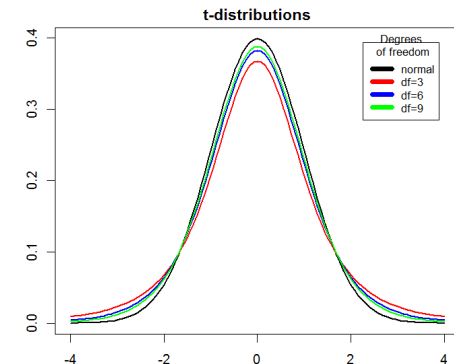
```
> # Add title
> title(main="t-distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title="Degrees\n of freedom", labelv,
+        cex=0.8, lwd=6, lty=1, col=colors)
```

Cauchy Distribution

The *Cauchy* distribution is Student's *t-distribution* with one degree of freedom $\nu = 1$, with the probability density function:

$$f(x) = \frac{1}{\pi\sigma} \frac{1}{((x-\mu)/\sigma)^2 + 1}$$

Where μ is the location parameter (equal to the mean) and σ is the scale parameter.

Since the *Cauchy* distribution has an infinite standard deviation, its measure of dispersion is the *interquartile range* (IQR), which is equal to σ.

The *interquartile range* is a *robust* measure of dispersion (scale), defined as the difference between the 75th minus the 25th percentiles.

The function dcauchy() calculates the *Cauchy* probability density.

The probability density of the *Cauchy* distribution decreases as the second power for large values of x:

$$f(x) \propto 1/x^2$$

Cauchy and Normal Distributions
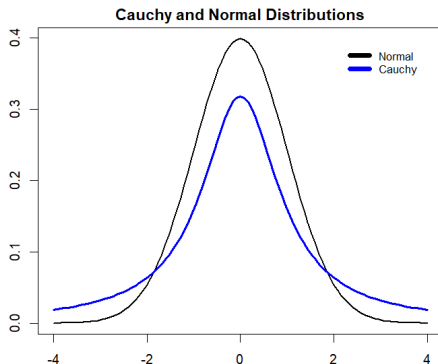

```
> # Plot the Normal and Cauchy probability distributions
> curve(expr=dnorm, xlim=c(-4, 4), xlab="", ylab="", lwd=2)
> curve(expr=dcauchy, lwd=3, col="blue", add=TRUE)
> # Add title
> title(main="Cauchy and Normal Distributions", line=0.5)
> # Add legend
> legend("topright", inset=0.05, bty="n",
+        title=NULL,leg=c("Normal", "Cauchy"),
+        cex=0.8, lwd=6, lty=1, col=c("black", "blue"))
```

Pareto Distribution and *Zipf's* Law

The probability density of Student's *t-distribution* decreases as a power for large values of x:

$$f(x) \propto |x|^{-(\nu+1)}$$

The probability density of the *Pareto* distribution decreases as a power of the random variable x:

$$f(x) = \alpha x^{-(\alpha+1)}$$

For $x > 1$ and decay parameter $\alpha > 1$.

The mean μ and variance σ^2 of the *Pareto* distribution are equal to:
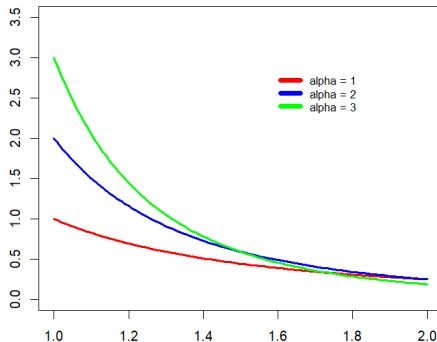
$$\mu = \frac{\alpha}{\alpha - 1} \qquad \sigma^2 = \frac{\alpha}{(\alpha - 1)^2(\alpha - 2)}$$

Zipf's law is analogous to the *Pareto* distribution, and applies to discrete variables.

Zipf's law states that the frequency f of a given value is inversely proportional to its rank n in the frequency table: $f(n) \propto n^{-s}$.

For example, *Zipf's law* applies to the frequency of words in a natural language.

Pareto Distributions


```
> # Define Pareto function
> pare_to <- function(x, alpha)
+   alpha*x^(-alpha-1)
> colors <- c("red", "blue", "green")
> alpha_s <- c(1.0, 2.0, 3.0)
> for (indeks in 1:3) {  # Plot three curves
+   curve(expr=pare_to(x, alpha_s[indeks]),
+   xlim=c(1, 2), ylim=c(0.0, 3.5),
+   xlab="", ylab="", lwd=3, col=colors[indeks],
+   add=as.logical(indeks-1))
+ }  # end for
> # Add title and legend
> title(main="Pareto Distributions", line=0.5)
> # labols <- pasto("alpha", 1:3, som=" ")
```

Poisson Probability Distribution

The *Poisson* distribution gives the probability of the number of events observed in an interval of space or time.

The *Poisson* probability function is given by:

$$f(n; \lambda) = \frac{\lambda^n \cdot e^{-\lambda}}{n!}$$

The *Poisson* random variable `n` is the number of events observed in the interval.

The parameter λ is the average number of events that are observed in the interval.

An example of a *Poisson* distribution is the number of mail items received each day.

Poisson distribution

```
> poisson_events <- 0:11  # Poisson events
> poisson_freq <- dpois(poisson_events, lambda=4)
> names(poisson_freq) <- as.character(poisson_events)
> # Poisson function
> poissonfun <- function(x, lambda)
+                {exp(-lambda)*lambda^x/factorial(x)}
> curve(expr=poissonfun(x, lambda=4), xlim=c(0, 11), main="Poisson distribution",
+ xlab="No. of events", ylab="Frequency of events", lwd=2, col="red")
> legend(x="topright", legend="Poisson density", title="", bty="n",
+  inset=0.05, cex=0.8, bg="white", lwd=4, lty=1, col="red")
```
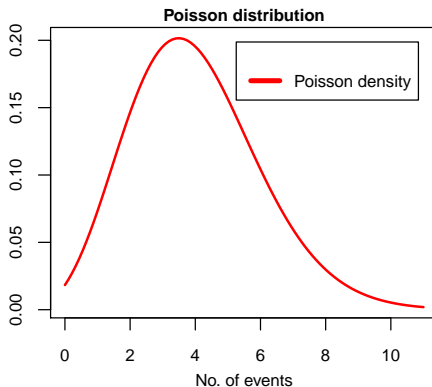
Homework Assignment

Required

- Study all the lecture slides in `FRE6871_Lecture_1.pdf`, and run all the code in `FRE6871_Lecture_1.R`,
- Study the *RStudio Style Guide*.

Recommended

- Read about the *Vasicek* single factor model in `Vasicek Portfolio Default Distribution.pdf`, `BOE Credit Risk Models.pdf`, `BIS Bank Capital Model.pdf`, and in `Elizalde CDO Vasicek Credit Model.pdf`.