

FREE eBook

LEARNING

R Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#r

www.dbooks.org

Table of Contents

About ... 1

Chapter 1: Getting started with R Language ... 2

Remarks ... 2

Editing R Docs on Stack Overflow ... 2

A few features of R that immigrants from other language may find unusual ... 2

Examples ... 2

Installing R ... 2

Windows only: ... 2

For Windows ... 2

For OSX / macOS ... 3

Alternative 1 ... 3

Alternative 2 ... 3

For Debian, Ubuntu and derivatives ... 3

For Red Hat and Fedora ... 3

For Archlinux ... 4

Hello World! ... 4

Getting Help ... 4

Interactive mode and R scripts ... 4

The interactive mode ... 4

Using R as a calculator ... 4

The first plot ... 6

R scripts ... 8

Chapter 2: *apply family of functions (functionals) ... 9

Remarks ... 9

Members of the *apply Family ... 9

Examples ... 9

Use anonymous functions with apply ... 10

Bulk File Loading ... 11

Combining multiple `data.frames` (`lapply`, `mapply`) ... 11

Using built-in functionals ... 13
Built-in functionals: lapply(), sapply(), and mapply() ... 13
lapply() ... 13
sapply() ... 13
mapply() ... 13
Using user-defined functionals ... 14
User-defined functionals ... 14
Chapter 3: .Rprofile ... 16
Remarks ... 16
Examples ... 16
.Rprofile - the first chunk of code executed ... 16
Setting your R home directory ... 16
Setting page size options ... 16
set the default help type ... 16
set a site library ... 16
Set a CRAN mirror ... 17
Setting the location of your library ... 17
Custom shortcuts or functions ... 17
Pre-loading the most useful packages ... 17
See Also ... 17
.Rprofile example ... 17
Startup ... 18
Options ... 18
Custom Functions ... 18
Chapter 4: Aggregating data frames ... 19
Introduction ... 19
Examples ... 19
Aggregating with base R ... 19
Aggregating with dplyr ... 20
Aggregating with data.table ... 21
Chapter 5: Analyze tweets with R ... 23

www.dbooks.org

Introduction ... 23
Examples ... 23
Download Tweets ... 23
R Libraries ... 23
Get text of tweets ... 24
Chapter 6: ANOVA ... 25
Examples ... 25
Basic usage of aov() ... 25
Basic usage of Anova() ... 25
Chapter 7: Arima Models ... 27
Remarks ... 27
Examples ... 27
Modeling an AR1 Process with Arima ... 27
Chapter 8: Arithmetic Operators ... 36
Remarks ... 36
Examples ... 36
Range and addition ... 36
Addition and subtraction ... 37
Chapter 9: Bar Chart ... 40
Introduction ... 40
Examples ... 40
barplot() function ... 40
Chapter 10: Base Plotting ... 48
Parameters ... 48
Remarks ... 48
Examples ... 48
Basic Plot ... 48
Matplot ... 51
Histograms ... 57
Combining Plots ... 59
par() ... 59

layout() 60
Density plot 61
Empirical Cumulative Distribution Function 63
Getting Started with R_Plots 64
Chapter 11: Bibliography in RMD 66
Parameters 66
Remarks 66
Examples 67
Specifying a bibliography and cite authors 67
Inline references 68
Citation styles 68
Chapter 12: boxplot 71
Syntax 71
Parameters 71
Examples 71
Create a box-and-whisker plot with boxplot() {graphics} 71
Simple boxplot (Sepal.Length) 72
Boxplot of sepal length grouped by species 72
Bring order 73
Change groups names 74
Small improvements 75
Color 75
Proximity of the box 76
See the summaries which the boxplots are based plot=FALSE 76
Additional boxplot style parameters 77
Box 77
Median 77
Whisker 77
Staple 77
Outliers 78
Example 78

www.dbooks.org

Chapter 13: caret ... 80
Introduction ... 80
Examples ... 80
Preprocessing ... 80
Chapter 14: Classes ... 82
Introduction ... 82
Remarks ... 82
Examples ... 82
Vectors ... 82
Inspect classes ... 82
Vectors and lists ... 83
Chapter 15: Cleaning data ... 85
Introduction ... 85
Examples ... 85
Removing missing data from a vector ... 85
Removing incomplete rows ... 85
Chapter 16: Code profiling ... 87
Examples ... 87
System.time ... 87
proc.time() ... 87
Line Profiling ... 88
Microbenchmark ... 89
Benchmarking using microbenchmark ... 90
Chapter 17: Coercion ... 92
Introduction ... 92
Examples ... 92
Implicit Coercion ... 92
Chapter 18: Color schemes for graphics ... 93
Examples ... 93
viridis - print and colorblind friendly palettes ... 93
RColorBrewer ... 96
A handy function to glimse a vector of colors ... 98

colorspace - click&drag interface for colors ... 99
basic R color functions ... 100
Colorblind-friendly palettes ... 101
Chapter 19: Column wise operation ... 104
Examples ... 104
sum of each column ... 104
Chapter 20: Combinatorics ... 106
Examples ... 106
Enumerating combinations of a specified length ... 106
Without replacement ... 106
With replacement ... 106
Counting combinations of a specified length ... 107
Without replacement ... 107
With replacement ... 107
Chapter 21: Control flow structures ... 108
Remarks ... 108
Optimizing Structure of For Loops ... 108
Vectorizing For Loops ... 109
Examples ... 110
Basic For Loop Construction ... 110
Optimal Construction of a For Loop ... 110
Poorly optimized for loop ... 111
Well optimized for loop ... 111
vapply Function ... 111
colMeans Function ... 111
Efficiency comparison ... 111
The Other Looping Constructs: while and repeat ... 112
The while loop ... 112
The repeat loop ... 113
More on break ... 113
Chapter 22: Creating packages with devtools ... 115

www.dbooks.org

Introduction 115
Remarks 115
Examples 115
Creating and distributing packages 115
Creation of the documentation 115
Construction of the package skeleton 116
Edition of the package properties 116
1. Package description 116
2. Optional folders 116
Finalization and build 117
Distribution of your package 117
Through Github 117
Through CRAN 117
Creating vignettes 117
Requirements 118
Vignette creation 118
Chapter 23: Creating reports with RMarkdown 119
Examples 119
Printing tables 119
Including LaTeX Preample Commands 121
Including bibliographies 122
Basic R-markdown document structure 123
R-markdown code chunks 123
R-markdown document example 123
Converting R-markdown to other formats 124
Chapter 24: Creating vectors 126
Examples 126
Sequence of numbers 126
seq() 126
Vectors 127
Creating named vectors 129

Expanding a vector with the rep() function ... 130
Vectors from build in constants: Sequences of letters & month names ... 131
Chapter 25: Data acquisition ... 133
Introduction ... 133
Examples ... 133
Built-in datasets ... 133
Example ... 133
Datasets within packages ... 134
Gapminder ... 134
World Population Prospects 2015 - United Nations Population Department ... 134
Packages to access open databases ... 134
Eurostat ... 134
Packages to access restricted data ... 136
Human Mortality Database ... 136
Chapter 26: Data frames ... 139
Syntax ... 139
Examples ... 139
Create an empty data.frame ... 139
Subsetting rows and columns from a data frame ... 140
Syntax for accessing rows and columns: [, [[, and $ ... 140
Like a matrix: data[rows, columns] ... 141
With numeric indexes ... 141
With column (and row) names ... 141
Rows and columns together ... 142
A warning about dimensions: ... 142
Like a list ... 142
With single brackets data[columns] ... 142
With double brackets data[[one_column]] ... 143
Using $ to access columns ... 143
Drawbacks of $ for accessing columns ... 143
Advanced indexing: negative and logical indices ... 144

www.dbooks.org

Negative indices omit elements 144
Logical vectors indicate specific elements to keep 144
Convenience functions to manipulate data.frames 145
subset 145
transform 145
with and within 145
Introduction 146
Convert data stored in a list to a single data frame using do.call 147
Convert all columns of a data.frame to character class 148
Subsetting Rows by Column Values 148
Chapter 27: data.table 150
Introduction 150
Syntax 150
Remarks 151
Installation and support 151
Loading the package 152
Examples 152
Creating a data.table 152
Build 152
Read in 152
Modify a data.frame 153
Coerce object to data.table 153
Adding and modifying columns 153
Editing entire columns 154
Editing subsets of columns 154
Editing column attributes 155
Special symbols in data.table 155
.SD 155
.SDcols 156
.N 156
Writing code compatible with both data.frame and data.table 157

Differences in subsetting syntax ... 157
Strategies for maintaining compatibility with data.frame and data.table ... 158
Setting keys in data.table ... 159
Chapter 28: Date and Time ... 162
Introduction ... 162
Remarks ... 162
Classes ... 162
Selecting a date-time format ... 162
Specialized packages ... 163
Examples ... 163
Current Date and Time ... 163
Go to the End of the Month ... 164
Go to First Day of the Month ... 164
Move a date a number of months consistently by months ... 164
Chapter 29: Date-time classes (POSIXct and POSIXlt) ... 166
Introduction ... 166
Remarks ... 166
Pitfalls ... 166
Related topics ... 166
Specialized packages ... 166
Examples ... 166
Formatting and printing date-time objects ... 166
Parsing strings into date-time objects ... 167
Notes ... 167
Missing elements ... 167
Time zones ... 167
Date-time arithmetic ... 168
Chapter 30: Debugging ... 169
Examples ... 169
Using browser ... 169
Using debug ... 170

www.dbooks.org

Chapter 31: Distribution Functions ... 171
Introduction ... 171
Remarks ... 171
Examples ... 171
Normal distribution ... 171
Binomial Distribution ... 172
Chapter 32: dplyr ... 176
Remarks ... 176
Examples ... 176
dplyr's single table verbs ... 176
Syntax commonalities ... 176
filter ... 177
arrange ... 178
select ... 179
mutate ... 180
summarise ... 181
group_by ... 181
Putting it all togther ... 182
summarise multiple columns ... 182
Subset Observation (Rows) ... 185
dplyr::filter() - Select a subset of rows in a data frame that meet a logical criteria: ... 185
dplyr::distinct() - Remove duplicate rows: ... 185
Aggregating with %>% (pipe) operator ... 186
Examples of NSE and string variables in dpylr ... 187
Chapter 33: Expression: parse + eval ... 188
Remarks ... 188
Examples ... 188
Execute code in string format ... 188
Chapter 34: Extracting and Listing Files in Compressed Archives ... 189
Examples ... 189
Extracting files from a .zip archive ... 189

Listing files in a .zip archive ... 189
Listing files in a .tar archive ... 189
Extracting files from a .tar archive ... 189
Extract all .zip archives in a directory ... 190
Chapter 35: Factors ... 191
Syntax ... 191
Remarks ... 191
Mapping the integer to the level ... 192
Modern use of factors ... 192
Examples ... 193
Basic creation of factors ... 193
Consolidating Factor Levels with a List ... 194
Consolidating levels using factor (factor_approach) ... 195
Consolidating levels using ifelse (ifelse_approach) ... 195
Consolidating Factors Levels with a List (list_approach) ... 195
Benchmarking each approach ... 196
Factors ... 196
Changing and reordering factors ... 197
Rebuilding factors from zero ... 202
Problem ... 202
Solution ... 203
Chapter 36: Fault-tolerant/resilient code ... 204
Parameters ... 204
Remarks ... 204
tryCatch ... 204
Implications of choosing specific return values of the handler functions ... 204
"Undesired" warning message ... 205
Examples ... 205
Using tryCatch() ... 205
Function definition using tryCatch ... 205
Testing things out ... 206
Investigating the output ... 207

www.dbooks.org

Chapter 37: Feature Selection in R -- Removing Extraneous Features 208
Examples 208
Removing features with zero or near-zero variance 208
Removing features with high numbers of NA 208
Removing closely correlated features 208
Chapter 38: Formula 210
Examples 210
The basics of formula 210
Create Linear, Quadratic and Second Order Interaction Terms 211
Chapter 39: Fourier Series and Transformations 213
Remarks 213
Examples 214
Fourier Series 214
Chapter 40: Functional programming 221
Examples 221
Built-in Higher Order Functions 221
Chapter 41: Generalized linear models 222
Examples 222
Logistic regression on Titanic dataset 222
Chapter 42: Get user input 225
Syntax 225
Examples 225
User input in R 225
Chapter 43: ggplot2 226
Remarks 226
Examples 226
Scatter Plots 226
Displaying multiple plots 227
Prepare your data for plotting 231
Add horizontal and vertical lines to plot 233
Add one common horizontal line for all categorical variables 233

Add one horizontal line for each categorical variable **233**

Add horizontal line over grouped bars **233**

Add vertical line **233**

Vertical and Horizontal Bar Chart 233

Violin plot 233

Produce basic plots with qplot 233

Chapter 44: GPU-accelerated computing **236**

Remarks 236

Examples 236

gpuR gpuMatrix objects 236

gpuR vclMatrix objects 236

Chapter 45: Hashmaps **238**

Examples 238

Environments as hash maps 238

Introduction 238

Insertion 238

Key Lookup 239

Inspecting the Hash Map 239

Flexibility 240

Limitations 241

package:hash 242

package:listenv 243

Chapter 46: heatmap and heatmap.2 ..... **244**

Examples 244

Examples from the official documentation 244

stats::heatmap **244**

Example 1 (Basic usage) 244

Example 2 (no column dendrogram (nor reordering) at all) 245

Example 3 ("no nothing") 245

Example 4 (with reorder()) 246

Example 5 (NO reorder()) 247

www.dbooks.org

Example 6 (slightly artificial with color bar, without ordering) 248
Example 7 (slightly artificial with color bar, with ordering) 249
Example 8 (For variable clustering, rather use distance based on cor()) 250
Tuning parameters in heatmap.2 252
Chapter 47: Hierarchical clustering with hclust 258
Introduction 258
Remarks 258
Examples 258
Example 1 - Basic use of hclust, display of dendrogram, plot clusters 258
Example 2 - hclust and outliers 262
Chapter 48: Hierarchical Linear Modeling 265
Examples 265
basic model fitting 265
Chapter 49: I/O for database tables 266
Remarks 266
Specialized packages 266
Examples 266
Reading Data from MySQL Databases 266
General 266
Using limits 266
Reading Data from MongoDB Databases 266
Chapter 50: I/O for foreign tables (Excel, SAS, SPSS, Stata) 268
Examples 268
Importing data with rio 268
Importing Excel files 268
Reading excel files with the xlsx package 269
Reading Excel files with the XLconnect package 269
Reading excel files with the openxlsx package 270
Reading excel files with the readxl package 270
Reading excel files with the RODBC package 271
Reading excel files with the gdata package 272

Read and write Stata, SPSS and SAS files ... 272
Import or Export of Feather file ... 273
Chapter 51: I/O for geographic data (shapefiles, etc.) ... 275
Introduction ... 275
Examples ... 275
Import and Export Shapefiles ... 275
Chapter 52: I/O for raster images ... 276
Introduction ... 276
Examples ... 276
Load a multilayer raster ... 276
Chapter 53: I/O for R's binary format ... 278
Examples ... 278
Rds and RData (Rda) files ... 278
Enviromments ... 278
Chapter 54: Implement State Machine Pattern using S4 Class ... 280
Introduction ... 280
Examples ... 280
Parsing Lines using State Machine ... 280
Chapter 55: Input and output ... 294
Remarks ... 294
Examples ... 294
Reading and writing data frames ... 294
Writing ... 294
Reading ... 294
Further resources ... 295
Chapter 56: Inspecting packages ... 296
Introduction ... 296
Remarks ... 296
Examples ... 296
View package information ... 296
View package's built-in data sets ... 296

www.dbooks.org

List a package's exported functions ... 296
View Package Version ... 296
View Loaded packages in Current Session ... 297
Chapter 57: Installing packages ... 298
Syntax ... 298
Parameters ... 298
Remarks ... 298
Related Docs ... 298
Examples ... 298
Download and install packages from repositories ... 298
Using CRAN ... 298
Using Bioconductor ... 299
Install package from local source ... 299
Install packages from GitHub ... 300
Using a CLI package manager -- basic pacman usage ... 301
Install local development version of a package ... 302
Chapter 58: Introduction to Geographical Maps ... 304
Introduction ... 304
Examples ... 304
Basic map-making with map() from the package maps ... 304
50 State Maps and Advanced Choropleths with Google Viz ... 308
Interactive plotly maps ... 309
Making Dynamic HTML Maps with Leaflet ... 311
Dynamic Leaflet maps in Shiny applications ... 313
Chapter 59: Introspection ... 315
Examples ... 315
Functions for Learning about Variables ... 315
Chapter 60: JSON ... 317
Examples ... 317
JSON to / from R objects ... 317
Chapter 61: Linear Models (Regression) ... 319

Syntax ... 319
Parameters ... 319
Examples ... 320
Linear regression on the mtcars dataset ... 320
Plotting The Regression (base) ... 321
Weighting ... 323
Checking for nonlinearity with polynomial regression ... 325
Quality assessment ... 328
Using the 'predict' function ... 329
Chapter 62: Lists ... 331
Examples ... 331
Quick Introduction to Lists ... 331
Introduction to lists ... 333
Reasons for using lists ... 333
Convert a list to a vector while keeping empty list elements ... 334
Serialization: using lists to pass informations ... 335
Chapter 63: lubridate ... 337
Syntax ... 337
Remarks ... 337
Examples ... 337
Parsing dates and datetimes from strings with lubridate ... 338
Dates ... 338
Datetimes ... 338
Utility functions ... 338
Parser functions ... 339
Parsing date and time in lubridate ... 339
Manipulating date and time in lubridate ... 340
Instants ... 340
Intervals, Durations and Periods ... 341
Rounding dates ... 342
Difference between period and duration ... 343
Time Zones ... 343

www.dbooks.org

Chapter 64: Machine learning 345
Examples 345
Creating a Random Forest model 345
Chapter 65: Matrices 346
Introduction 346
Examples 346
Creating matrices 346
Chapter 66: Meta: Documentation Guidelines 348
Remarks 348
Examples 348
Making good examples 348
Style 348
Prompts 348
Console output 348
Assignment 348
Code comments 349
Sections 349
Chapter 67: Missing values 350
Introduction 350
Remarks 350
Examples 350
Examining missing data 350
Reading and writing data with NA values 350
Using NAs of different classes 351
TRUE/FALSE and/or NA 351
Omitting or replacing missing values 352
Recoding missing values 352
Removing missing values 353
Excluding missing values from calculations 353
Chapter 68: Modifying strings by substitution 354
Introduction 354

Examples ... 354
Rearrange character strings using capture groups ... 354
Eliminate duplicated consecutive elements ... 354
Chapter 69: Natural language processing ... 356
Introduction ... 356
Examples ... 356
Create a term frequency matrix ... 356
Chapter 70: Network analysis with the igraph package ... 358
Examples ... 358
Simple Directed and Non-directed Network Graphing ... 358
Chapter 71: Non-standard evaluation and standard evaluation ... 360
Introduction ... 360
Examples ... 360
Examples with standard dplyr verbs ... 360
Chapter 72: Numeric classes and storage modes ... 362
Examples ... 362
Numeric ... 362
Chapter 73: Object-Oriented Programming in R ... 364
Introduction ... 364
Examples ... 364
S3 ... 364
Chapter 74: Parallel processing ... 366
Remarks ... 366
Examples ... 366
Parallel processing with foreach package ... 366
Parallel processing with parallel package ... 367
Random Number Generation ... 368
mcparallelDo ... 369
Example ... 369
Other Examples ... 369
Chapter 75: Pattern Matching and Replacement ... 371

www.dbooks.org

Introduction ... 371
Syntax ... 371
Remarks ... 371
Differences from other languages ... 371
Specialized packages ... 371
Examples ... 371
Making substitutions ... 371
Finding Matches ... 372
Is there a match? ... 372
Match locations ... 372
Matched values ... 372
Details ... 373
Summary of matches ... 373
Single and Global match ... 373
Find matches in big data sets ... 375
Chapter 76: Performing a Permutation Test ... 376
Examples ... 376
A fairly general function ... 376
Chapter 77: Pipe operators (%>% and others) ... 379
Introduction ... 379
Syntax ... 379
Parameters ... 379
Remarks ... 379
Packages that use %>% ... 379
Finding documentation ... 380
Hotkeys ... 380
Performance Considerations ... 380
Examples ... 380
Basic use and chaining ... 380
Functional sequences ... 381
Assignment with %<>% ... 382

Exposing contents with %$% ... 382
Using the pipe with dplyr and ggplot2 ... 383
Creating side effects with %T>% ... 383
Chapter 78: Pivot and unpivot with data.table ... 385
Syntax ... 385
Parameters ... 385
Remarks ... 385
Examples ... 385
Pivot and unpivot tabular data with data.table - I ... 385
Pivot and unpivot tabular data with data.table - II ... 387
Chapter 79: Probability Distributions with R ... 389
Examples ... 389
PDF and PMF for different distributions in R ... 389
Chapter 80: Publishing ... 390
Introduction ... 390
Remarks ... 390
Examples ... 390
Formatting tables ... 390
Printing to plain text ... 390
Printing delimited tables ... 390
Further resources ... 390
Formatting entire documents ... 391
Further Resources ... 391
Chapter 81: R code vectorization best practices ... 392
Examples ... 392
By row operations ... 392
Chapter 82: R in LaTeX with knitr ... 396
Syntax ... 396
Parameters ... 396
Remarks ... 396
Examples ... 397

www.dbooks.org

R in Latex with Knitr and Code Externalization ... 397
R in Latex with Knitr and Inline Code Chunks ... 398
R in LaTex with Knitr and Internal Code Chunks ... 398
Chapter 83: R Markdown Notebooks (from RStudio) ... 399
Introduction ... 399
Examples ... 399
Creating a Notebook ... 399
Inserting Chunks ... 400
Executing Chunk Code ... 401
Splitting Code into Chunks ... 401
Execution Progress ... 402
Executing Multiple Chunks ... 403
Preview Output ... 404
Saving and Sharing ... 404
Chapter 84: R memento by examples ... 406
Introduction ... 406
Examples ... 406
Data types ... 406
Vectors ... 406
Matrices ... 406
Dataframes ... 406
Lists ... 406
Environments ... 407
Plotting (using plot) ... 407
Commonly used functions ... 407
Chapter 85: Random Forest Algorithm ... 409
Introduction ... 409
Examples ... 409
Basic examples - Classification and Regression ... 409
Chapter 86: Random Numbers Generator ... 411
Examples ... 411

Random permutations 411
Random number generator's reproducibility 411
Generating random numbers using various density functions 412
Uniform distribution between 0 and 10 412
Normal distribution with 0 mean and standard deviation of 1 412
Binomial distribution with 10 trials and success probability of 0.5 412
Geometric distribution with 0.2 success probability 412
Hypergeometric distribution with 3 white balls, 10 black balls and 5 draws 413
Negative Binomial distribution with 10 trials and success probability of 0.8 413
Poisson distribution with mean and variance (lambda) of 2 413
Exponential distribution with the rate of 1.5 413
Logistic distribution with 0 location and scale of 1 413
Chi-squared distribution with 15 degrees of freedom 413
Beta distribution with shape parameters a=1 and b=0.5 413
Gamma distribution with shape parameter of 3 and scale=0.5 413
Cauchy distribution with 0 location and scale of 1 414
Log-normal distribution with 0 mean and standard deviation of 1 (on log scale) 414
Weibull distribution with shape parameter of 0.5 and scale of 1 414
Wilcoxon distribution with 10 observations in the first sample and 20 in second. 414
Multinomial distribution with 5 object and 3 boxes using the specified probabilities 414
Chapter 87: Randomization 415
Introduction 415
Remarks 415
Examples 415
Random draws and permutations 415
Random permutation 415
Draws without Replacement 416
Draws with Replacement 416
Changing Draw Probabilities 417
Setting the seed 418
Chapter 88: Raster and Image Analysis 419

www.dbooks.org

Introduction ... 419
Examples ... 419
Calculating GLCM Texture ... 419
Mathematical Morphologies ... 421
Chapter 89: Rcpp ... 424
Examples ... 424
Inline Code Compile ... 424
Rcpp Attributes ... 424
Extending Rcpp with Plugins ... 425
Specifying Additional Build Dependencies ... 426
Chapter 90: Reading and writing strings ... 427
Remarks ... 427
Examples ... 427
Printing and displaying strings ... 427
Reading from or writing to a file connection ... 429
Capture output of operating system command ... 429
Functions which return a character vector ... 429
Functions which return a data frame ... 430
Chapter 91: Reading and writing tabular data in plain-text files (CSV, TSV, etc.) ... 431
Syntax ... 431
Parameters ... 431
Remarks ... 431
Examples ... 432
Importing .csv files ... 432
Importing using base R ... 432
Notes ... 432
Importing using packages ... 432
Importing with data.table ... 433
Notes ... 434
Importing .tsv files as matrices (basic R) ... 434
Exporting .csv files ... 435

Exporting using base R ... 435
Exporting using packages ... 435
Import multiple csv files ... 435
Importing fixed-width files ... 435
Importing with base R ... 436
Importing with readr ... 436
Chapter 92: Recycling ... 437
Remarks ... 437
Examples ... 437
Recycling use in subsetting ... 437
Chapter 93: Regular Expression Syntax in R ... 439
Introduction ... 439
Examples ... 439
Use `grep` to find a string in a character vector ... 439
Chapter 94: Regular Expressions (regex) ... 441
Introduction ... 441
Remarks ... 441
Character classes ... 441
Quantifiers ... 441
Start and end of line indicators ... 441
Differences from other languages ... 441
Additional Resources ... 442
Examples ... 442
Eliminating Whitespace ... 442
Trimming Whitespace ... 442
Removing Leading Whitespace ... 442
Removing Trailing Whitespace ... 443
Removing All Whitespace ... 443
Validate a date in a "YYYYMMDD" format ... 443
Validate US States postal abbreviations ... 444
Validate US phone numbers ... 444

www.dbooks.org

Escaping characters in R regex patterns ... 445
Differences between Perl and POSIX regex ... 446
Look-ahead/look-behind ... 446
Chapter 95: Reproducible R ... 447
Introduction ... 447
Remarks ... 447
References ... 447
Examples ... 447
Data reproducibility ... 447
dput() and dget() ... 447
Package reproducibility ... 448
Chapter 96: Reshape using tidyr ... 449
Introduction ... 449
Examples ... 449
Reshape from long to wide format with spread() ... 449
Reshape from wide to long format with gather() ... 450
h21 ... 450
Chapter 97: Reshaping data between long and wide forms ... 451
Introduction ... 451
Remarks ... 451
Helpful packages ... 451
Examples ... 451
The reshape function ... 451
Long to Wide ... 452
Wide to Long ... 452
Reshaping data ... 453
Base R ... 453
The tidyr package ... 454
The data.table package ... 454
Chapter 98: RESTful R Services ... 455
Introduction ... 455

Examples ... 455
opencpu Apps ... 455
Chapter 99: RMarkdown and knitr presentation ... 456
Syntax ... 456
Parameters ... 456
Remarks ... 456
Sub options parameters: ... 456
Examples ... 460
Rstudio example ... 460
Adding a footer to an ioslides presentation ... 460
Chapter 100: RODBC ... 463
Examples ... 463
Connecting to Excel Files via RODBC ... 463
SQL Server Management Database connection to get individual table ... 463
Connecting to relational databases ... 463
Chapter 101: roxygen2 ... 464
Parameters ... 464
Examples ... 464
Documenting a package with roxygen2 ... 464
Writing with roxygen2 ... 464
Building the documentation ... 465
Chapter 102: Run-length encoding ... 466
Remarks ... 466
Extensions ... 466
Examples ... 466
Run-length Encoding with `rle` ... 466
Identifying and grouping by runs in base R ... 467
Identifying and grouping by runs in data.table ... 467
Run-length encoding to compress and decompress vectors ... 468
Chapter 103: Scope of variables ... 470
Remarks ... 470

www.dbooks.org

Examples ... 470
Environments and Functions ... 470
Sub functions ... 471
Global Assignment ... 471
Explicit Assignment of Environments and Variables ... 472
Function Exit ... 472
Packages and Masking ... 473
Chapter 104: Set operations ... 474
Remarks ... 474
Examples ... 474
Set operators for pairs of vectors ... 474
Comparing sets ... 474
Combining sets ... 474
Set membership for vectors ... 475
Cartesian or "cross" products of vectors ... 475
Applying functions to combinations ... 475
Make unique / drop duplicates / select distinct elements from a vector ... 476
Measuring set overlaps / Venn diagrams for vectors ... 476
Chapter 105: Shiny ... 478
Examples ... 478
Create an app ... 478
One file ... 478
Two files ... 478
Create ui.R file ... 478
Create server.R file ... 479
Radio Button ... 479
Checkbox Group ... 479
Select box ... 480
Launch a Shiny app ... 481
1. Two files app ... 481
2. One file app ... 482

Control widgets ... 482
Debugging ... 484
Showcase mode ... 484
Reactive Log Visualizer ... 484
Chapter 106: Solving ODEs in R ... 485
Syntax ... 485
Parameters ... 485
Remarks ... 485
Examples ... 485
The Lorenz model ... 485
Lotka-Volterra or: Prey vs. predator ... 487
ODEs in compiled languages - definition in R ... 488
ODEs in compiled languages - definition in C ... 489
ODEs in compiled languages - definition in fortran ... 490
ODEs in compiled languages - a benchmark test ... 492
Chapter 107: Spark API (SparkR) ... 494
Remarks ... 494
Examples ... 494
Setup Spark context ... 494
Setup Spark context in R ... 494
Get Spark Cluster ... 494
Cache data ... 494
Create RDDs (Resilient Distributed Datasets) ... 495
From dataframe: ... 495
From csv: ... 495
Chapter 108: spatial analysis ... 496
Examples ... 496
Create spatial points from XY data set ... 496
Importing a shape file (.shp) ... 497
rgdal ... 497
raster ... 497

www.dbooks.org

tmap ... 498
Chapter 109: Speeding up tough-to-vectorize code ... 499
Examples ... 499
Speeding tough-to-vectorize for loops with Rcpp ... 499
Speeding tough-to-vectorize for loops by byte compiling ... 500
Chapter 110: Split function ... 502
Examples ... 502
Basic usage of split ... 502
Using split in the split-apply-combine paradigm ... 504
Chapter 111: sqldf ... 506
Examples ... 506
Basic Usage Examples ... 506
Chapter 112: Standardize analyses by writing standalone R scripts ... 508
Introduction ... 508
Remarks ... 508
Examples ... 508
The basic structure of standalone R program and how to call it ... 508
The first standalone R script ... 508
Preparing a standalone R script ... 509
Linux/Mac ... 509
Windows ... 509
Using littler to execute R scripts ... 510
Installing littler ... 510
From R: ... 510
Using apt-get (Debian, Ubuntu): ... 510
Using littler with standard .r scripts ... 510
Using littler on shebanged scripts ... 511
Chapter 113: String manipulation with stringi package ... 512
Remarks ... 512
Examples ... 512
Count pattern inside string ... 512

Duplicating strings ... 513
Paste vectors ... 513
Splitting text by some fixed pattern ... 513
Chapter 114: strsplit function ... 515
Syntax ... 515
Examples ... 515
Introduction ... 515
Chapter 115: Subsetting ... 517
Introduction ... 517
Remarks ... 517
Examples ... 518
Atomic vectors ... 518
Lists ... 519
Matrices ... 521
Selecting individual matrix entries by their positions ... 522
Data frames ... 523
Other objects ... 524
Vector indexing ... 525
Elementwise Matrix Operations ... 525
Some Functions used with Matrices ... 526
Chapter 116: Survival analysis ... 528
Examples ... 528
Random Forest Survival Analysis with randomForestSRC ... 528
Introduction - basic fitting and plotting of parametric survival models with the survival ... 529
Kaplan Meier estimates of survival curves and risk set tables with survminer ... 530
Chapter 117: Text mining ... 533
Examples ... 533
Scraping Data to build N-gram Word Clouds ... 533
Chapter 118: The character class ... 537
Introduction ... 537
Remarks ... 537
Related topics ... 537

www.dbooks.org

Examples ... 537
Coercion ... 537
Chapter 119: The Date class ... 538
Remarks ... 538
Related topics ... 538
Jumbled notes ... 538
More notes ... 538
Examples ... 538
Formatting Dates ... 538
Dates ... 539
Parsing Strings into Date Objects ... 541
Chapter 120: The logical class ... 542
Introduction ... 542
Remarks ... 542
Shorthand ... 542
Examples ... 542
Logical operators ... 542
Coercion ... 543
Interpretation of NAs ... 543
Chapter 121: tidyverse ... 544
Examples ... 544
Creating tbl_df's ... 544
tidyverse: an overview ... 544
What is tidyverse? ... 544
How to use it? ... 545
What are those packages? ... 545
Chapter 122: Time Series and Forecasting ... 547
Remarks ... 547
Examples ... 547
Exploratory Data Analysis with time-series data ... 547
Creating a ts object ... 548

Chapter 123: Updating R and the package library 550
Examples 550
On Windows 550
Chapter 124: Updating R version 551
Introduction 551
Examples 551
Installing from R Website 551
Updating from within R using installr Package 551
Deciding on the old packages 552
Updating Packages 555
Check R Version 556
Chapter 125: Using pipe assignment in your own package %<>%: How to ? 557
Introduction 557
Examples 557
Putting the pipe in a utility-functions file 557
Chapter 126: Using texreg to export models in a paper-ready way 558
Introduction 558
Remarks 558
Links 558
Examples 558
Printing linear regression results 558
Chapter 127: Variables 560
Examples 560
Variables, data structures and basic Operations 560
Types of data structures 561
Common operations and some cautionary advice 561
Example objects 561
Some vector operations 562
Some vector operation Warnings! 562
Some Matrix operations Warning! 562
"Private" variables 563

www.dbooks.org

Chapter 128: Web Crawling in R 564
Examples 564
Standard scraping approach using the RCurl package 564
Chapter 129: Web scraping and parsing 565
Remarks 565
Legality 565
Examples 565
Basic scraping with rvest 565
Using rvest when login is required 566
Chapter 130: Writing functions in R 568
Examples 568
Named functions 568
Anonymous functions 569
RStudio code snippets 569
Passing column names as argument of a function 570
Chapter 131: xgboost 572
Examples 572
Cross Validation and Tuning with xgboost 572
Credits 575

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: r-language

It is an unofficial and free R Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official R Language.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

www.dbooks.org

Chapter 1: Getting started with R Language

Remarks

Editing R Docs on Stack Overflow

See the documentation guidelines for general rules when creating documentation.

A few features of R that immigrants from other language may find unusual

- Unlike other languages variables in R need not require type declaration.
- The same variable can be assigned different data types at different instances of time, if required.
- Indexing of atomic vectors and lists starts from 1, not 0.
- R `arrays` (and the special case of matrices) have a `dim` attribute that sets them apart from R's "atomic vectors" which have no attributes.
- A list in R allows you to gather a variety of objects under one name (that is, the name of the list) in an ordered way. These objects can be **matrices**, **vectors**, **data frames**, **even other lists**, etc. It is not even required that these objects are related to each other in any way.
- Recycling
- Missing values

Examples

Installing R

You might wish to install RStudio after you have installed R. RStudio is a development environment for R that simplifies many programming tasks.

Windows only:

Visual Studio (starting from version 2015 Update 3) now features a development environment for R called R Tools, that includes a live interpreter, IntelliSense, and a debugging module. If you choose this method, you won't have to install R as specified in the following section.

For Windows

1. Go to the CRAN website, click on download R for Windows, and download the latest version

of R.
2. Right-click the installer file and RUN as administrator.
3. Select the operational language for installation.
4. Follow the instructions for installation.

For OSX / macOS

Alternative 1

(0. Ensure XQuartz is installed)

1. Go to the CRAN website and download the latest version of R.
2. Open the disk image and run the installer.
3. Follow the instructions for installation.

This will install both R and the R-MacGUI. It will put the GUI in the /Applications/ Folder as R.app where it can either be double-clicked or dragged to the Doc. When a new version is released, the (re)-installation process will overwrite R.app but prior major versions of R will be maintained. The actual R code will be in the /Library/Frameworks/R.Framework/Versions/ directory. Using R within RStudio is also possible and would be using the same R code with a different GUI.

Alternative 2

1. Install homebrew (the missing package manager for macOS) by following the instructions on https://brew.sh/
2. `brew install R`

Those choosing the second method should be aware that the maintainer of the Mac fork advises against it, and will not respond to questions about difficulties on the R-SIG-Mac Mailing List.

For Debian, Ubuntu and derivatives

You can get the version of R corresponding to your distro via `apt-get`. However, this version will frequently be quite far behind the most recent version available on CRAN. You can add CRAN to your list of recognized "sources".

```
sudo apt-get install r-base
```

You can get a more recent version directly from CRAN by adding CRAN to your sources list. Follow the directions from CRAN for more details. Note in particular the need to also execute this so that you can use `install.packages()`. Linux packages are usually distributed as source files and need compilation:

```
sudo apt-get install r-base-dev
```


www.dbooks.org

For Red Hat and Fedora

```
sudo dnf install R
```

For Archlinux

R is directly available in the `Extra` package repo.

```
sudo pacman -S r
```

More info on using R under Archlinux can be found on the ArchWiki R page.

Hello World!

```
"Hello World!"
```

Also, check out the detailed discussion of how, when, whether and why to print a string.

Getting Help

You can use function `help()` or `?` to access documentations and search for help in R. For even more general searches, you can use `help.search()` or `??`.

```
#For help on the help function of R
help()

#For help on the paste function
help(paste)     #OR
help("paste")   #OR
?paste          #OR
?"paste"
```

Visit https://www.r-project.org/help.html for additional information

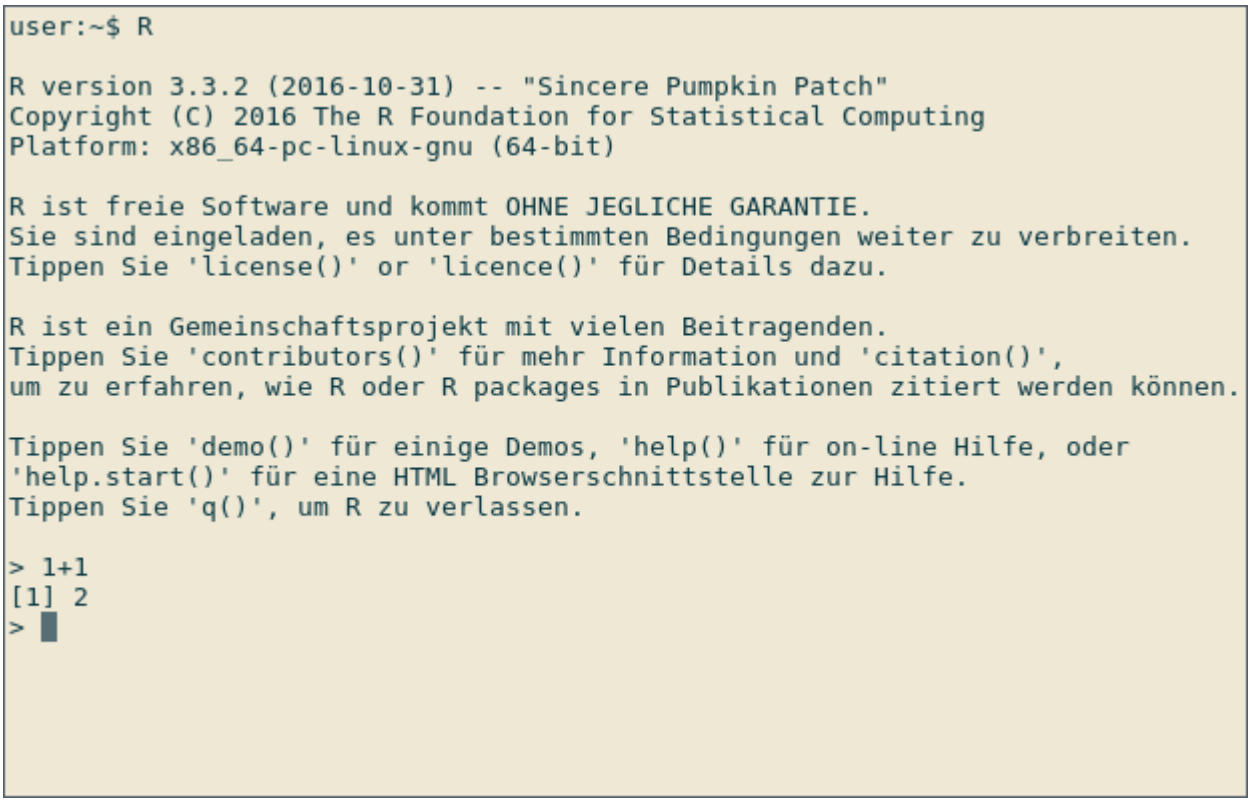
Interactive mode and R scripts

The interactive mode

The most basic way to use R is the *interactive* mode. You type commands and immediately get the result from R.

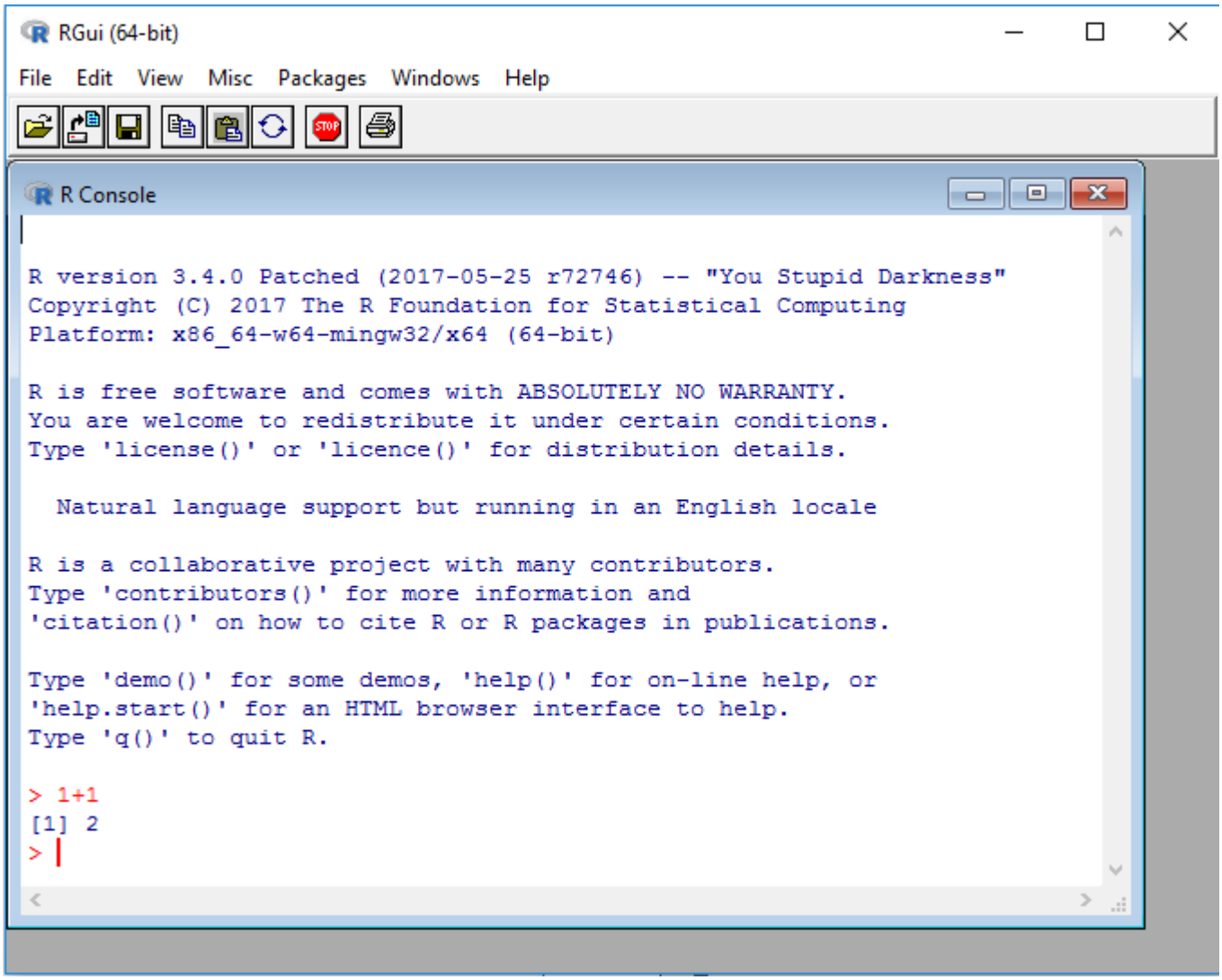
Using R as a calculator

Start R by typing `R` at the command prompt of your operating system or by executing `RGui` on Windows. Below you can see a screenshot of an interactive R session on Linux:



This is RGui on Windows, the most basic working environment for R under Windows:

www.dbooks.org



After the `>` sign, expressions can be typed in. Once an expression is typed, the result is shown by R. In the screenshot above, R is used as a calculator: Type

```
1+1
```

to immediately see the result, `2`. The leading `[1]` indicates that R returns a vector. In this case, the vector contains only one number (2).

The first plot

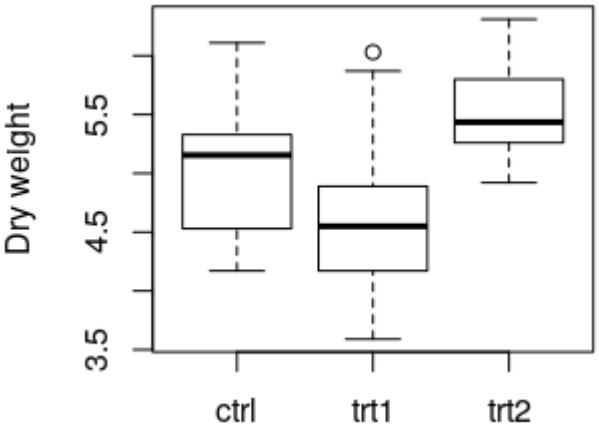
R can be used to generate plots. The following example uses the data set `PlantGrowth`, which comes as an example data set along with R

Type int the following all lines into the R prompt which do not start with `##`. Lines starting with `##` are meant to document the result which R will return.

```
data(PlantGrowth)
str(PlantGrowth)
## 'data.frame':    30 obs. of  2 variables:
## $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
## $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
anova(lm(weight ~ group, data = PlantGrowth))
## Analysis of Variance Table
##
## Response: weight
##           Df  Sum Sq Mean Sq F value  Pr(>F)
## group      2  3.7663  1.8832  4.8461 0.01591 *
## Residuals 27 10.4921  0.3886
## ---
## Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
boxplot(weight ~ group, data = PlantGrowth, ylab = "Dry weight")
```

The following plot is created:



`data(PlantGrowth)` loads the example data set `PlantGrowth`, which is records of dry masses of plants which were subject to two different treatment conditions or no treatment at all (control group). The data set is made available under the name `PlantGrowth`. Such a name is also called a Variable.

To load your own data, the following two documentation pages might be helpful:

- Reading and writing tabular data in plain-text files (CSV, TSV, etc.)
- I/O for foreign tables (Excel, SAS, SPSS, Stata)

`str(PlantGrowth)` shows information about the data set which was loaded. The output indicates that `PlantGrowth` is a `data.frame`, which is R's name for a table. The `data.frame` contains of two columns and 30 rows. In this case, each row corresponds to one plant. Details of the two columns are shown in the lines starting with `$`: The first column is called `weight` and contains numbers (`num`, the dry weight of the respective plant). The second column, `group`, contains the treatment that the plant was subjected to. This is categorial data, which is called `factor` in R. Read more information about data frames.

To compare the dry masses of the three different groups, a one-way ANOVA is performed using `anova(lm( ... )).weight ~ group` means "Compare the values of the column `weight`, grouping by the values of the column `group`". This is called a Formula in R. `data = ...` specifies the name of the table where the data can be found.

The result shows, among others, that there exists a significant difference (Column `Pr(>F)`), `p = 0.01591`) between some of the three groups. Post-hoc tests, like Tukey's Test, must be performed to determine which groups' means differ significantly.

www.dbooks.org

`boxplot(...)` creates a box plot of the data. where the values to be plotted come from. `weight ~ group` means: "Plot the values of the column weight *versus* the values of the column `group`. `ylab = ...` specifies the label of the y axis. More information: Base plotting

Type `q()` or `Ctrl-D` to exit from the R session.

R scripts

To document your research, it is favourable to save the commands you use for calculation in a file. For that effect, you can create **R scripts**. An R script is a simple text file, containing R commands.

Create a text file with the name `plants.R`, and fill it with the following text, where some commands are familiar from the code block above:

```
data(PlantGrowth)

anova(lm(weight ~ group, data = PlantGrowth))

png("plant_boxplot.png", width = 400, height = 300)
boxplot(weight ~ group, data = PlantGrowth, ylab = "Dry weight")
dev.off()
```

Execute the script by typing into your terminal (The terminal of your operating system, **not** an interactive R session like in the previous section!)

```
R --no-save <plant.R >plant_result.txt
```

The file `plant_result.txt` contains the results of your calculation, as if you had typed them into the interactive R prompt. Thereby, your calculations are documented.

The new commands `png` and `dev.off` are used for saving the boxplot to disk. The two commands must enclose the plotting command, as shown in the example above. `png("FILENAME", width = ..., height = ...)` opens a new PNG file with the specified file name, width and height in pixels. `dev.off()` will finish plotting and saves the plot to disk. No output is saved until `dev.off()` is called.

Read Getting started with R Language online: https://riptutorial.com/r/topic/360/getting-started-with-r-language

Chapter 2: *apply family of functions (functionals)

Remarks

A function in the `*apply` family is an abstraction of a `for` loop. Compared with the `for` loops `*apply` functions have the following advantages:

1. Require less code to write.
2. Doesn't have an iteration counter.
3. Doesn't use temporary variables to store intermediate results.

However `for` loops are more general and can give us more control allowing to achieve complex computations that are not always trivial to do using `*apply` functions.

The relationship between `for` loops and `*apply` functions is explained in the documentation for `for` loops.

Members of the `*apply` Family

The `*apply` family of functions contains several variants of the same principle that differ based primarily on the kind of output they return.

function	Input	Output
`apply`	`matrix`, `data.frame`, or `array`	vector or matrix (depending on the length of each element returned)
`sapply`	vector or `list`	vector or matrix (depending on the length of each element returned)
`lapply`	vector or `list`	`list`
`vapply`	vector or `list	vector or matrix (depending on the length of each element returned) of the user-designated class
`mapply`	multiple vectors, `lists` or a combination	`list`

See "Examples" to see how each of these functions is used.

Examples

www.dbooks.org

Use anonymous functions with apply

`apply` is used to evaluate a function (maybe an anonymous one) over the margins of an array or matrix.

Let's use the `iris` dataset to illustrate this idea. The `iris` dataset has measurements of 150 flowers from 3 species. Let's see how this dataset is structured:

```
> head(iris)

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Now, imagine that you want to know the mean of *each* of these variables. One way to solve this might be to use a `for` loop, but R programmers will often prefer to use `apply` (for reasons why, see Remarks):

```
> apply(iris[1:4], 2, mean)

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

- In the first parameter, we subset `iris` to include only the first 4 columns, because `mean` only works on numeric data.
- The second parameter value of `2` indicates that we want to work on the columns only (the second subscript of the r×c array); `1` would give the row means.

In the same way we can calculate more meaningful values:

```
# standard deviation
apply(iris[1:4], 2, sd)
# variance
apply(iris[1:4], 2, var)
```

Caveat: R has some built-in functions which are better for calculating column and row sums and means: `colMeans` and `rowMeans`.

Now, let's do a different and more meaningful task: let's calculate the mean *only* for those values which are bigger than `0.5`. For that, we will create our own `mean` function.

```
> our.mean.function <- function(x) { mean(x[x > 0.5]) }
> apply(iris[1:4], 2, our.mean.function)

Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.665347
```

(Note the difference in the mean of `Petal.Width`)

But, what if we don't want to use this function in the rest of our code? Then, we can use an anonymous function, and write our code like this:

```
apply(iris[1:4], 2, function(x) { mean(x[x > 0.5]) })
```

So, as we have seen, we can use `apply` to execute the same operation on columns or rows of a dataset using only one line.

Caveat: Since `apply` returns very different kinds of output depending on the length of the results of the specified function, it may not be the best choice in cases where you are not working interactively. Some of the other `*apply` family functions are a bit more predictable (see Remarks).

Bulk File Loading

for a large number of files which may need to be operated on in a similar process and with well structured file names.

firstly a vector of the file names to be accessed must be created, there are multiple options for this:

- Creating the vector manually with `paste0()`

  ```
  files <- paste0("file_", 1:100, ".rds")
  ```

- Using `list.files()` with a regex search term for the file type, requires knowledge of regular expressions (regex) if other files of same type are in the directory.

  ```
  files <- list.files("./", pattern = "\\.rds$", full.names = TRUE)
  ```

where `x` is a vector of part of the files naming format used.

`lapply` will output each response as element of a list.

`readRDS` is specific to `.rds` files and will change depending on the application of the process.

```
my_file_list <- lapply(files, readRDS)
```

This is not necessarily faster than a for loop from testing but allows all files to be an element of a list without assigning them explicitly.

Finally, we often need to load multiple packages at once. This trick can do it quite easily by applying `library()` to all libraries that we wish to import:

```
lapply(c("jsonlite","stringr","igraph"),library,character.only=TRUE)
```

Combining multiple `data.frames` (`lapply`, `mapply`)

www.dbooks.org

In this exercise, we will generate four bootstrap linear regression models and combine the summaries of these models into a single data frame.

```
library(broom)

#* Create the bootstrap data sets
BootData <- lapply(1:4,
                   function(i) mtcars[sample(1:nrow(mtcars),
                                             size = nrow(mtcars),
                                             replace = TRUE), ])

#* Fit the models
Models <- lapply(BootData,
                 function(BD) lm(mpg ~ qsec + wt + factor(am),
                                 data = BD))

#* Tidy the output into a data.frame
Tidied <- lapply(Models,
                 tidy)

#* Give each element in the Tidied list a name
Tidied <- setNames(Tidied, paste0("Boot", seq_along(Tidied)))
```

At this point, we can take two approaches to inserting the names into the data.frame.

```
#* Insert the element name into the summary with `lapply`
#* Requires passing the names attribute to `lapply` and referencing `Tidied` within
#* the applied function.
Described_lapply <-
 lapply(names(Tidied),
        function(nm) cbind(nm, Tidied[[nm]]))

Combined_lapply <- do.call("rbind", Described_lapply)

#* Insert the element name into the summary with `mapply`
#* Allows us to pass the names and the elements as separate arguments.
Described_mapply <-
 mapply(
  function(nm, dframe) cbind(nm, dframe),
  names(Tidied),
  Tidied,
  SIMPLIFY = FALSE)

Combined_mapply <- do.call("rbind", Described_mapply)
```

If you're a fan of `magrittr` style pipes, you can accomplish the entire task in a single chain (though it may not be prudent to do so if you need any of the intermediary objects, such as the model objects themselves):

```
library(magrittr)
library(broom)
Combined <- lapply(1:4,
                   function(i) mtcars[sample(1:nrow(mtcars),
                                             size = nrow(mtcars),
                                             replace = TRUE), ]) %>%
 lapply(function(BD) lm( mpg ~ qsec + wt + factor(am), data = BD)) %>%
 lapply(tidy) %>%
```

```
 setNames(paste0("Boot", seq_along(.))) %>%
 mapply(function(nm, dframe) cbind(nm, dframe),
        nm = names(.),
        dframe = .,
        SIMPLIFY = FALSE) %>%
 do.call("rbind", .)
```

Using built-in functionals

Built-in functionals: lapply(), sapply(), and mapply()

R comes with built-in functionals, of which perhaps the most well-known are the apply family of functions. Here is a description of some of the most common apply functions:

- `lapply()` = takes a list as an argument and applies the specified function to the list.
- `sapply()` = the same as `lapply()` but attempts to simplify the output to a vector or a matrix.
    - `vapply()` = a variant of `sapply()` in which the output object's type must be specified.
- `mapply()` = like `lapply()` but can pass multiple vectors as input to the specified function. Can be simplified like `sapply()`.
    - `Map()` is an alias to `mapply()` with `SIMPLIFY = FALSE`.

---

lapply()

`lapply()` can be used with two different iterations:

- `lapply(variable, FUN)`
- `lapply(seq_along(variable), FUN)`

```
# Two ways of finding the mean of x
set.seed(1)
df <- data.frame(x = rnorm(25), y = rnorm(25))
lapply(df, mean)
lapply(seq_along(df), function(x) mean(df[[x]]))
```

---

sapply()

`sapply()` will attempt to resolve its output to either a vector or a matrix.

```
# Two examples to show the different outputs of sapply()
sapply(letters, print)  ## produces a vector
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
sapply(x, quantile)  ## produces a matrix
```

---

mapply()

www.dbooks.org

`mapply()` works much like `lapply()` except it can take multiple vectors as input (hence the m for multivariate).

```
mapply(sum, 1:5, 10:6, 3) # 3 will be "recycled" by mapply
```

Using user-defined functionals

User-defined functionals

Users can create their own functionals to varying degrees of complexity. The following examples are from Functionals by Hadley Wickham:

```
randomise <- function(f) f(runif(1e3))

lapply2 <- function(x, f, ...) {
    out <- vector("list", length(x))
    for (i in seq_along(x)) {
        out[[i]] <- f(x[[i]], ...)
    }
    out
}
```

In the first case, `randomise` accepts a single argument `f`, and calls it on a sample of Uniform random variables. To demonstrate equivalence, we call `set.seed` below:

```
set.seed(123)
randomise(mean)
#[1] 0.4972778

set.seed(123)
mean(runif(1e3))
#[1] 0.4972778


set.seed(123)
randomise(max)
#[1] 0.9994045

set.seed(123)
max(runif(1e3))
#[1] 0.9994045
```

The second example is a re-implementation of `base::lapply`, which uses functionals to apply an operation (`f`) to each element in a list (`x`). The `...` parameter allows the user to pass additional arguments to `f`, such as the `na.rm` option in the `mean` function:

```
lapply(list(c(1, 3, 5), c(2, NA, 6)), mean)
# [[1]]
# [1] 3
#
# [[2]]
# [1] NA
```

```
lapply2(list(c(1, 3, 5), c(2, NA, 6)), mean)
# [[1]]
# [1] 3
#
# [[2]]
# [1] NA


lapply(list(c(1, 3, 5), c(2, NA, 6)), mean, na.rm = TRUE)
# [[1]]
# [1] 3
#
# [[2]]
# [1] 4

lapply2(list(c(1, 3, 5), c(2, NA, 6)), mean, na.rm = TRUE)
# [[1]]
# [1] 3
#
# [[2]]
# [1] 4
```

Read *apply family of functions (functionals) online: https://riptutorial.com/r/topic/3567/-apply-family-of-functions--functionals-

www.dbooks.org

Chapter 3: .Rprofile

Remarks

There is a nice chapter on the matter in Efficient R programming

Examples

.Rprofile - the first chunk of code executed

`.Rprofile` is a file containing R code that is executed when you launch R from the directory containing the `.Rprofile` file. The similarly named `Rprofile.site`, located in R's home directory, is executed by default every time you load R from any directory. `Rprofile.site` and to a greater extend `.Rprofile` can be used to initialize an R session with personal preferences and various utility functions that you have defined.

> Important note: if you use RStudio, you can have a separate `.Rprofile` in every RStudio project directory.

Here are some examples of code that you might include in an .Rprofile file.

Setting your R home directory

```
# set R_home
Sys.setenv(R_USER="c:/R_home") # just an example directory
# but don't confuse this with the $R_HOME environment variable.
```

Setting page size options

```
options(papersize="a4")
options(editor="notepad")
options(pager="internal")
```

set the default help type

```
options(help_type="html")
```

set a site library

```
.Library.site <- file.path(chartr("\\", "/", R.home()), "site-library")
```

Set a CRAN mirror

```
local({r <- getOption("repos")
    r["CRAN"] <- "http://my.local.cran"
    options(repos=r)})
```

Setting the location of your library

This will allow you to not have to install all the packages again with each R version update.

```
# library location
.libPaths("c:/R_home/Rpackages/win")
```

Custom shortcuts or functions

Sometimes it is useful to have a shortcut for a long R expression. A common example of this setting an active binding to access the last top-level expression result without having to type out `.Last.value`:

```
makeActiveBinding(".", function(){.Last.value}, .GlobalEnv)
```

Because .Rprofile is just an R file, it can contain any arbitrary R code.

Pre-loading the most useful packages

This is bad practice and should generally be avoided because it separates package loading code from the scripts where those packages are actually used.

See Also

See `help(Startup)` for all the different startup scripts, and further aspects. In particular, two system-wide `Profile` files can be loaded as well. The first, `Rprofile`, may contain global settings, the other file `Profile.site` may contain local choices the system administrator can make for all users. Both files are found in the `${RHOME}/etc` directory of the R installation. This directory also contains global files `Renviron` and `Renviron.site` which both can be completemented with a local file `~/.Renviron` in the user's home directory.

.Rprofile example

www.dbooks.org

Startup

```
# Load library setwidth on start - to set the width automatically.
.First <- function() {
  library(setwidth)
  # If 256 color terminal - use library colorout.
  if (Sys.getenv("TERM") %in% c("xterm-256color", "screen-256color")) {
    library("colorout")
  }
}
```

Options

```
# Select default CRAN mirror for package installation.
options(repos=c(CRAN="https://cran.gis-lab.info/"))

# Print maximum 1000 elements.
options(max.print=1000)

# No scientific notation.
options(scipen=10)

# No graphics in menus.
options(menu.graphics=FALSE)

# Auto-completion for package names.
utils::rc.settings(ipck=TRUE)
```

Custom Functions

```
# Invisible environment to mask defined functions
.env = new.env()

# Quit R without asking to save.
.env$q <- function (save="no", ...) {
  quit(save=save, ...)
}

# Attach the environment to enable functions.
attach(.env, warn.conflicts=FALSE)
```

Read .Rprofile online: https://riptutorial.com/r/topic/4166/-rprofile

Chapter 4: Aggregating data frames

Introduction

Aggregation is one of the most common uses for R. There are several ways to do so in R, which we will illustrate here.

Examples

Aggregating with base R

For this, we will use the function aggregate, which can be used as follows:

```
aggregate(formula,function,data)
```

The following code shows various ways of using the aggregate function.

CODE:

```
df = data.frame(group=c("Group 1","Group 1","Group 2","Group 2","Group 2"), subgroup =
c("A","A","A","A","B"),value = c(2,2.5,1,2,1.5))

# sum, grouping by one column
aggregate(value~group, FUN=sum, data=df)

# mean, grouping by one column
aggregate(value~group, FUN=mean, data=df)

# sum, grouping by multiple columns
aggregate(value~group+subgroup,FUN=sum,data=df)

# custom function, grouping by one column
# in this example we want the sum of all values larger than 2 per group.
aggregate(value~group, FUN=function(x) sum(x[x>2]), data=df)
```

OUTPUT:

```
> df = data.frame(group=c("Group 1","Group 1","Group 2","Group 2","Group 2"), subgroup =
c("A","A","A","A","B"),value = c(2,2.5,1,2,1.5))
> print(df)
    group subgroup value
1 Group 1        A   2.0
2 Group 1        A   2.5
3 Group 2        A   1.0
4 Group 2        A   2.0
5 Group 2        B   1.5
>
> # sum, grouping by one column
> aggregate(value~group, FUN=sum, data=df)
    group value
1 Group 1   4.5
```


www.dbooks.org

```
2 Group 2   4.5
>
> # mean, grouping by one column
> aggregate(value~group, FUN=mean, data=df)
    group value
1 Group 1  2.25
2 Group 2  1.50
>
> # sum, grouping by multiple columns
> aggregate(value~group+subgroup,FUN=sum,data=df)
    group subgroup value
1 Group 1        A   4.5
2 Group 2        A   3.0
3 Group 2        B   1.5
>
> # custom function, grouping by one column
> # in this example we want the sum of all values larger than 2 per group.
> aggregate(value~group, FUN=function(x) sum(x[x>2]), data=df)
    group value
1 Group 1   2.5
2 Group 2   0.0
```

Aggregating with dplyr

Aggregating with dplyr is easy! You can use the group_by() and the summarize() functions for this. Some examples are given below.

CODE:

```
# Aggregating with dplyr
library(dplyr)

df = data.frame(group=c("Group 1","Group 1","Group 2","Group 2","Group 2"), subgroup =
c("A","A","A","A","B"),value = c(2,2.5,1,2,1.5))
print(df)

# sum, grouping by one column
df %>% group_by(group) %>% summarize(value = sum(value)) %>% as.data.frame()

# mean, grouping by one column
df %>% group_by(group) %>% summarize(value = mean(value)) %>% as.data.frame()

# sum, grouping by multiple columns
df %>% group_by(group,subgroup) %>% summarize(value = sum(value)) %>% as.data.frame()

# custom function, grouping by one column
# in this example we want the sum of all values larger than 2 per group.
df %>% group_by(group) %>% summarize(value = sum(value[value>2])) %>% as.data.frame()
```

OUTPUT:

```
> library(dplyr)
>
> df = data.frame(group=c("Group 1","Group 1","Group 2","Group 2","Group 2"), subgroup =
c("A","A","A","A","B"),value = c(2,2.5,1,2,1.5))
> print(df)
    group subgroup value
```

```
1 Group 1        A   2.0
2 Group 1        A   2.5
3 Group 2        A   1.0
4 Group 2        A   2.0
5 Group 2        B   1.5
>
> # sum, grouping by one column
> df %>% group_by(group) %>% summarize(value = sum(value)) %>% as.data.frame()
    group value
1 Group 1   4.5
2 Group 2   4.5
>
> # mean, grouping by one column
> df %>% group_by(group) %>% summarize(value = mean(value)) %>% as.data.frame()
    group value
1 Group 1  2.25
2 Group 2  1.50
>
> # sum, grouping by multiple columns
> df %>% group_by(group,subgroup) %>% summarize(value = sum(value)) %>% as.data.frame()
    group subgroup value
1 Group 1        A   4.5
2 Group 2        A   3.0
3 Group 2        B   1.5
>
> # custom function, grouping by one column
> # in this example we want the sum of all values larger than 2 per group.
> df %>% group_by(group) %>% summarize(value = sum(value[value>2])) %>% as.data.frame()
    group value
1 Group 1   2.5
2 Group 2   0.0
```

Aggregating with data.table

Grouping with the data.table package is done using the syntax `dt[i, j, by]` Which can be read out loud as: "*Take dt, subset rows using i, then calculate j, grouped by by.*" Within the dt statement, multiple calculations or groups should be put in a list. Since an alias for `list()` is `.()`, both can be used interchangeably. In the examples below we use `.()`.

CODE:

```
# Aggregating with data.table
library(data.table)

dt = data.table(group=c("Group 1","Group 1","Group 2","Group 2","Group 2"), subgroup =
c("A","A","A","A","B"),value = c(2,2.5,1,2,1.5))
print(dt)

# sum, grouping by one column
dt[,.(value=sum(value)),group]

# mean, grouping by one column
dt[,.(value=mean(value)),group]

# sum, grouping by multiple columns
dt[,.(value=sum(value)),.(group,subgroup)]

# custom function, grouping by one column
```


www.dbooks.org

```
# in this example we want the sum of all values larger than 2 per group.
dt[,.(value=sum(value[value>2])),group]
```

OUTPUT:

```
> # Aggregating with data.table
> library(data.table)
>
> dt = data.table(group=c("Group 1","Group 1","Group 2","Group 2","Group 2"), subgroup =
c("A","A","A","A","B"),value = c(2,2.5,1,2,1.5))
> print(dt)
     group subgroup value
1: Group 1        A   2.0
2: Group 1        A   2.5
3: Group 2        A   1.0
4: Group 2        A   2.0
5: Group 2        B   1.5
>
> # sum, grouping by one column
> dt[,.(value=sum(value)),group]
     group value
1: Group 1   4.5
2: Group 2   4.5
>
> # mean, grouping by one column
> dt[,.(value=mean(value)),group]
     group value
1: Group 1  2.25
2: Group 2  1.50
>
> # sum, grouping by multiple columns
> dt[,.(value=sum(value)),.(group,subgroup)]
     group subgroup value
1: Group 1        A   4.5
2: Group 2        A   3.0
3: Group 2        B   1.5
>
> # custom function, grouping by one column
> # in this example we want the sum of all values larger than 2 per group.
> dt[,.(value=sum(value[value>2])),group]
     group value
1: Group 1   2.5
2: Group 2   0.0
```

Read Aggregating data frames online: https://riptutorial.com/r/topic/10792/aggregating-data-frames

Chapter 5: Analyze tweets with R

Introduction

(Optional) Every topic has a focus. Tell the readers what they will find here and let future contributors know what belongs.

Examples

Download Tweets

The first think you need to do is to download tweets. You need to Setup your tweeter account. Much Information can be found in Internet on how to do it. The following two links were useful for my Setup (last checked in May 2017)

In particular I found the following two links useful (last checked in May 2017):

Link 1

Link 2

R Libraries

You will need the following R packages

```
library("devtools")
library("twitteR")
library("ROAuth")
```

Supposing you have your keys You have to run the following code

```
api_key <- XXXXXXXXXXXXXXXXXXXXX
api_secret <- XXXXXXXXXXXXXXXXXXXXX
access_token <- XXXXXXXXXXXXXXXXXXXXX
access_token_secret <- XXXXXXXXXXXXXXXXXXXXX


setup_twitter_oauth(api_key,api_secret)
```

Change `xxxxxxxxxxxxxxxxxxxxx` to your keys (if you have Setup your tweeter account you know which keys I mean).

Let's now suppose we want to download tweets on coffee. The following code will do it

```
search.string <- "#coffee"
no.of.tweets <- 1000
```


www.dbooks.org

```
c_tweets <- searchTwitter(search.string, n=no.of.tweets, lang="en")
```

You will get 1000 tweets on "coffee".

Get text of tweets

Now we need to access the text of the tweets. So we do it in this way (we also need to clean up the tweets from special characters that for now we don't need, like emoticons with the sapply function.)

```
coffee_tweets = sapply(c_tweets, function(t) t$getText())

coffee_tweets <- sapply(coffee_tweets,function(row) iconv(row, "latin1", "ASCII", sub=""))
```

and you can check your tweets with the `head` function.

```
head(coffee_tweets)
```

Read Analyze tweets with R online: https://riptutorial.com/r/topic/10086/analyze-tweets-with-r

Chapter 6: ANOVA

Examples

Basic usage of aov()

Analysis of Variance (aov) is used to determine if the means of two or more groups differ significantly from each other. Responses are assumed to be independent of each other, Normally distributed (within each group), and the within-group variances are assumed equal.

In order to complete the analysis data must be in long format (see reshaping data topic). `aov()` is a wrapper around the `lm()` function, using Wilkinson-Rogers formula notation `y~f` where `y` is the response (independent) variable and `f` is a factor (categorical) variable representing group membership. *If `f` is numeric rather than a factor variable, `aov()` will report the results of a linear regression in ANOVA format, which may surprise inexperienced users.*

The `aov()` function uses Type I (sequential) Sum of Squares. This type of Sum of Squares tests all of the (main and interaction) effects sequentially. The result is that the first effect tested is also assigned shared variance between it and other effects in the model. For the results from such a model to be reliable, data should be balanced (all groups are of the same size).

When the assumptions for Type I Sum of Squares do not hold, Type II or Type III Sum of Squares may be applicable. Type II Sum of Squares test each main effect after every other main effect, and thus controls for any overlapping variance. However, Type II Sum of Squares assumes no interaction between the main effects.

Lastly, Type III Sum of Squares tests each main effect after every other main effect *and* every interaction. This makes Type III Sum of Squares a necessity when an interaction is present.

Type II and Type III Sums of Squares are implemented in the `Anova()` function.

Using the `mtcars` data set as an example.

```
mtCarsAnovaModel <- aov(wt ~ factor(cyl), data=mtcars)
```

To view summary of ANOVA model:

```
summary(mtCarsAnovaModel)
```

One can also extract the coefficients of the underlying `lm()` model:

```
coefficients(mtCarsAnovaModel)
```

Basic usage of Anova()

www.dbooks.org

When dealing with an unbalanced design and/or non-orthogonal contrasts, Type II or Type III Sum of Squares are necessary. The `Anova()` function from the `car` package implements these. Type II Sum of Squares assumes no interaction between main effects. If interactions are assumed, Type III Sum of Squares is appropriate.

The `Anova()` function wraps around the `lm()` function.

Using the `mtcars` data sets as an example, demonstrating the difference between Type II and Type III when an interaction is tested.

```
> Anova(lm(wt ~ factor(cyl)*factor(am), data=mtcars), type = 2)
Anova Table (Type II tests)

Response: wt
                      Sum Sq Df F value    Pr(>F)
factor(cyl)           7.2278  2 11.5266 0.0002606 ***
factor(am)            3.2845  1 10.4758 0.0032895 **
factor(cyl):factor(am) 0.0668  2  0.1065 0.8993714
Residuals             8.1517 26
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

> Anova(lm(wt ~ factor(cyl)*factor(am), data=mtcars), type = 3)
Anova Table (Type III tests)

Response: wt
                       Sum Sq Df F value    Pr(>F)
(Intercept)           25.8427  1 82.4254 1.524e-09 ***
factor(cyl)            4.0124  2  6.3988  0.005498 **
factor(am)             1.7389  1  5.5463  0.026346 *
factor(cyl):factor(am) 0.0668  2  0.1065  0.899371
Residuals              8.1517 26
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Read ANOVA online: https://riptutorial.com/r/topic/3610/anova

Chapter 7: Arima Models

Remarks

The `Arima` function in the forecast package is more explicit in how it deals with constants, which may make it easier for some users relative to the `arima` function in base R.

ARIMA is a general framework for modeling and making predictions from time series data using (primarily) the series itself. The purpose of the framework is to differentiate short- and long-term dynamics in a series to improve the accuracy and certainty of forecasts. More poetically, ARIMA models provide a method for describing how shocks to a system transmit through time.

From an econometric perspective, ARIMA elements are necessary to correct serial correlation and ensure stationarity.

Examples

Modeling an AR1 Process with Arima

We will model the process

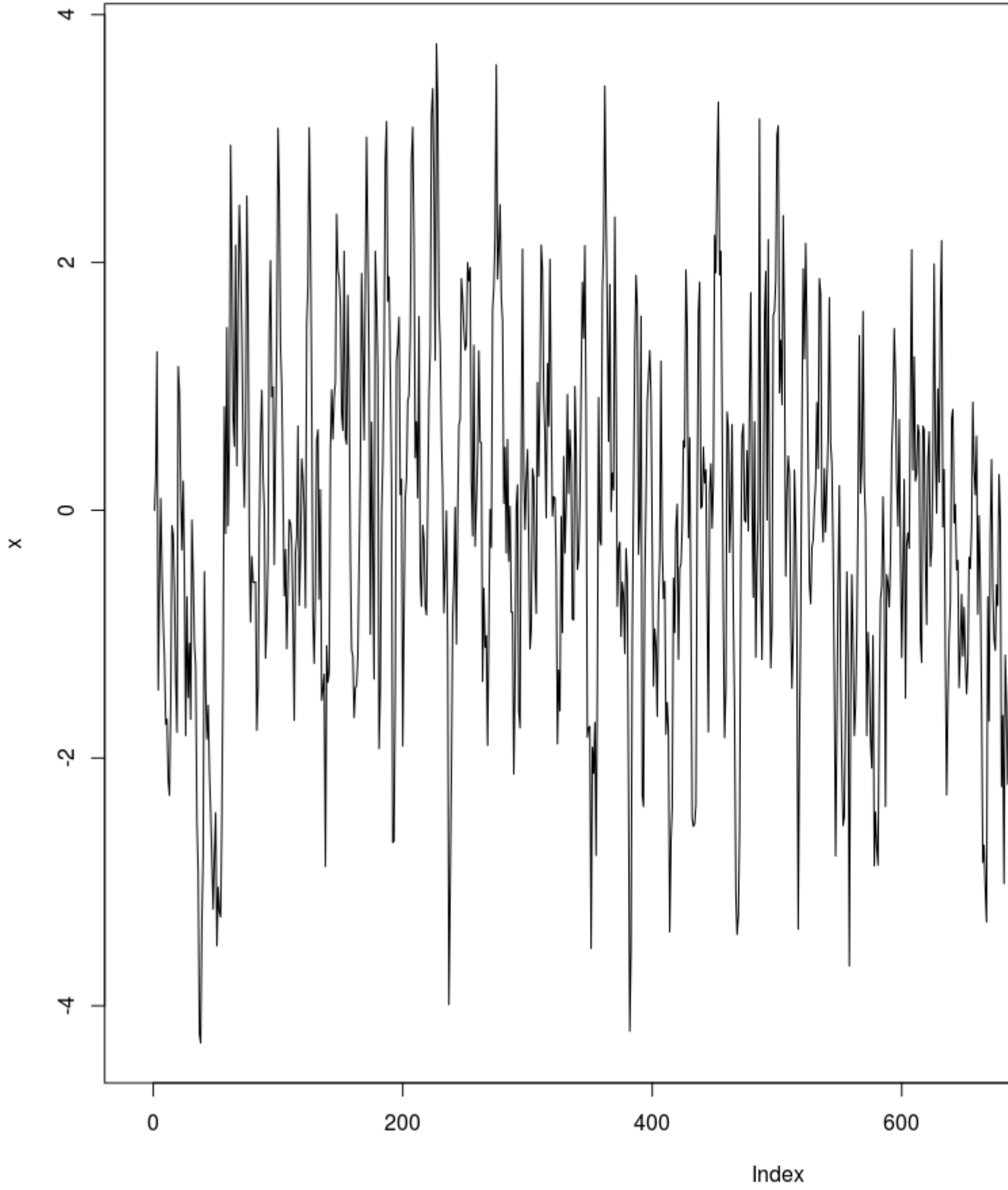
$$x_t = .7x_{t-1} + \epsilon \qquad \epsilon \sim N(0,1)$$

```
#Load the forecast package
library(forecast)

#Generate an AR1 process of length n (from Cowpertwait & Meltcalfe)
# Set up variables
set.seed(1234)
n <- 1000
x <- matrix(0,1000,1)
w <- rnorm(n)

# loop to create x
for (t in 2:n) x[t] <- 0.7 * x[t-1] + w[t]
plot(x,type='l')
```


www.dbooks.org


x
4
2
0
-2
-4
0
200
400
600
Index

We will fit an Arima model with autoregressive order 1, 0 degrees of differencing, and an MA order of 0.

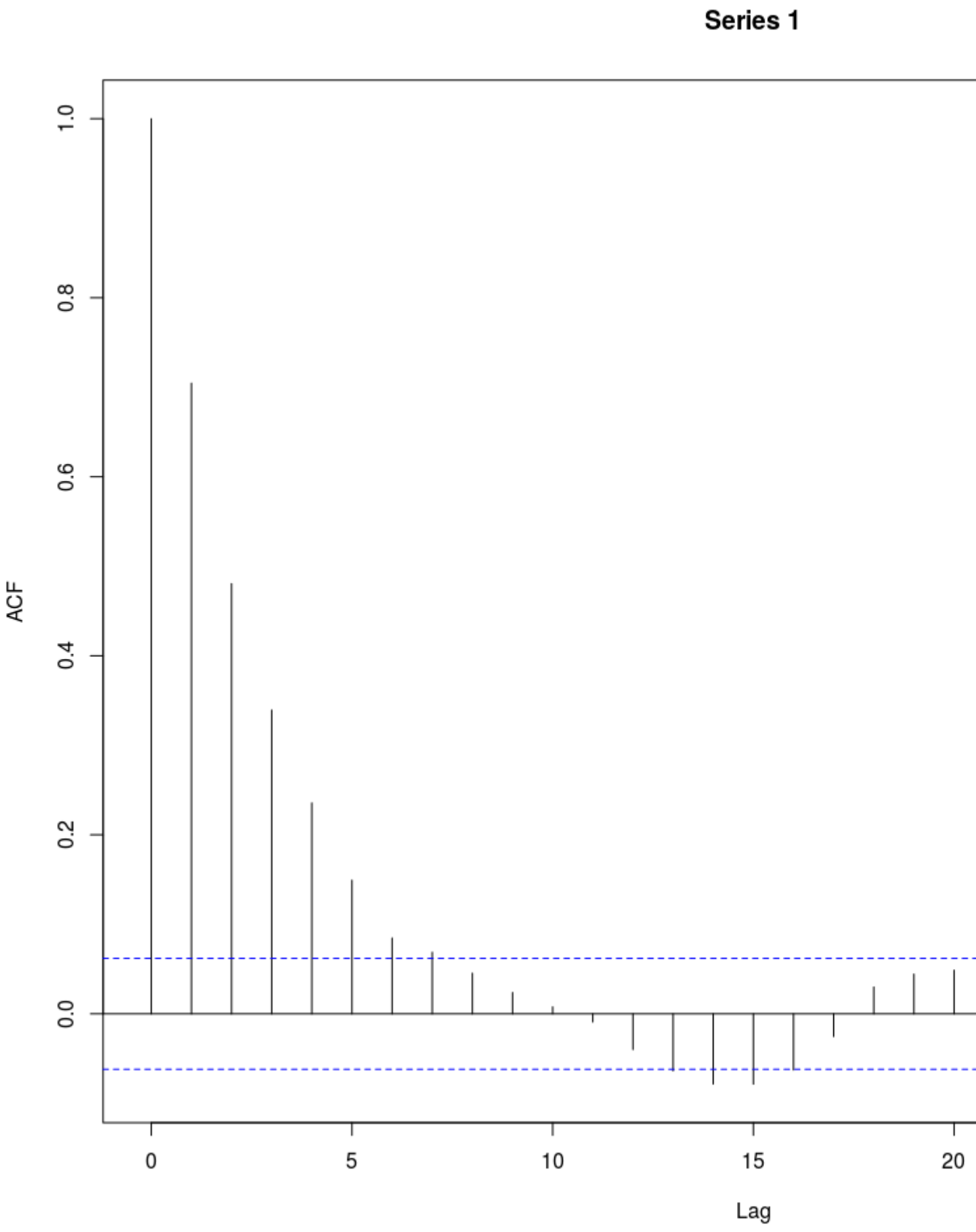
```
#Fit an AR1 model using Arima
fit <- Arima(x, order = c(1, 0, 0))
summary(fit)
# Series: x
# ARIMA(1,0,0) with non-zero mean
#
# Coefficients:
#          ar1  intercept
#       0.7040    -0.0842
# s.e.  0.0224     0.1062
#
# sigma^2 estimated as 0.9923:  log likelihood=-1415.39
# AIC=2836.79   AICc=2836.81   BIC=2851.51
#
# Training set error measures:
#                         ME      RMSE       MAE MPE MAPE    MASE       ACF1
# Training set -8.369365e-05 0.9961194 0.7835914 Inf  Inf 0.91488 0.02263595
# Verify that the model captured the true AR parameter
```

Notice that our coefficient is close to the true value from the generated data

```
fit$coef[1]
#       ar1
# 0.7040085

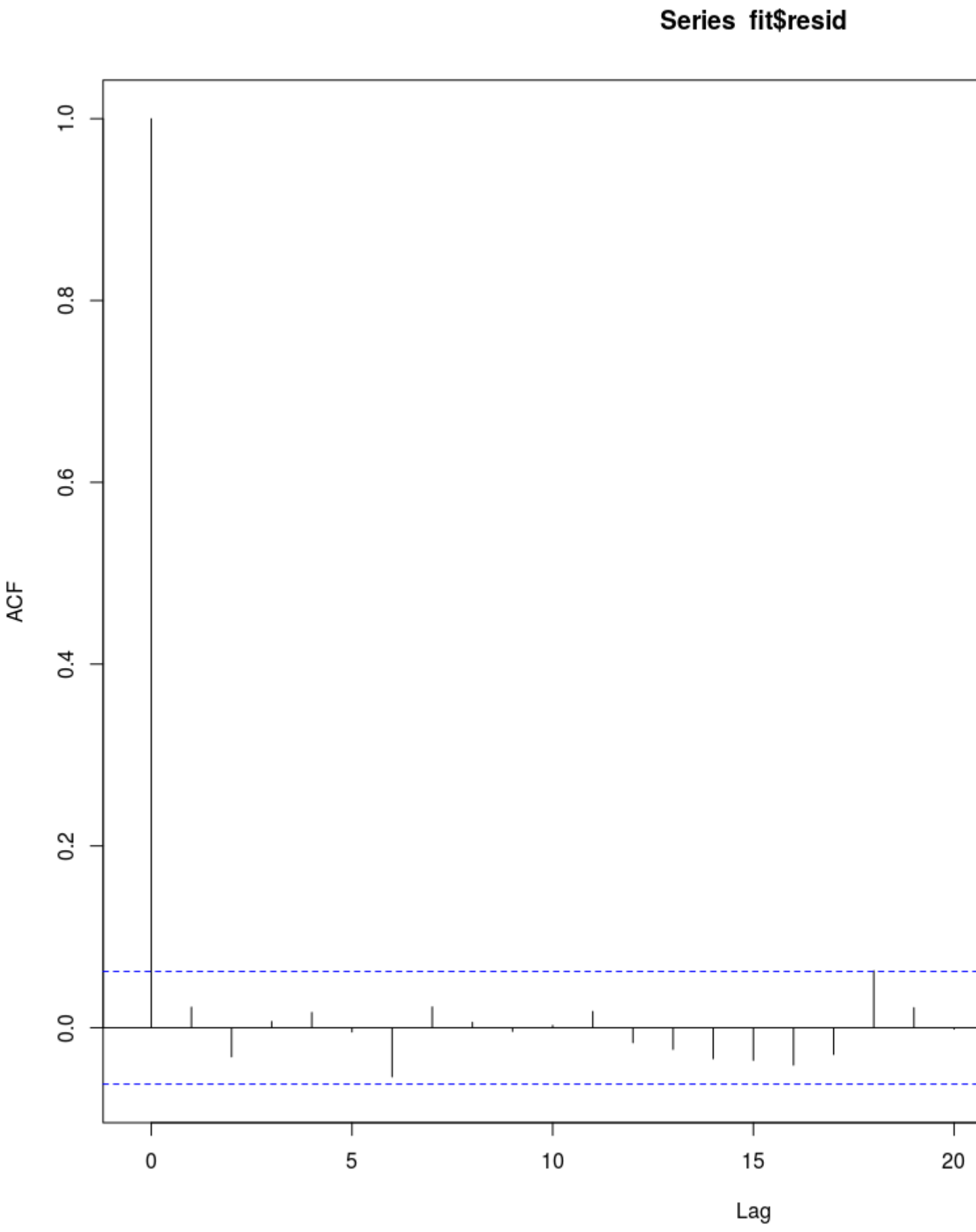
#Verify that the model eliminates the autocorrelation
acf(x)
```


www.dbooks.org


Series 1
ACF
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Lag

```
acf(fit$resid)
```


www.dbooks.org

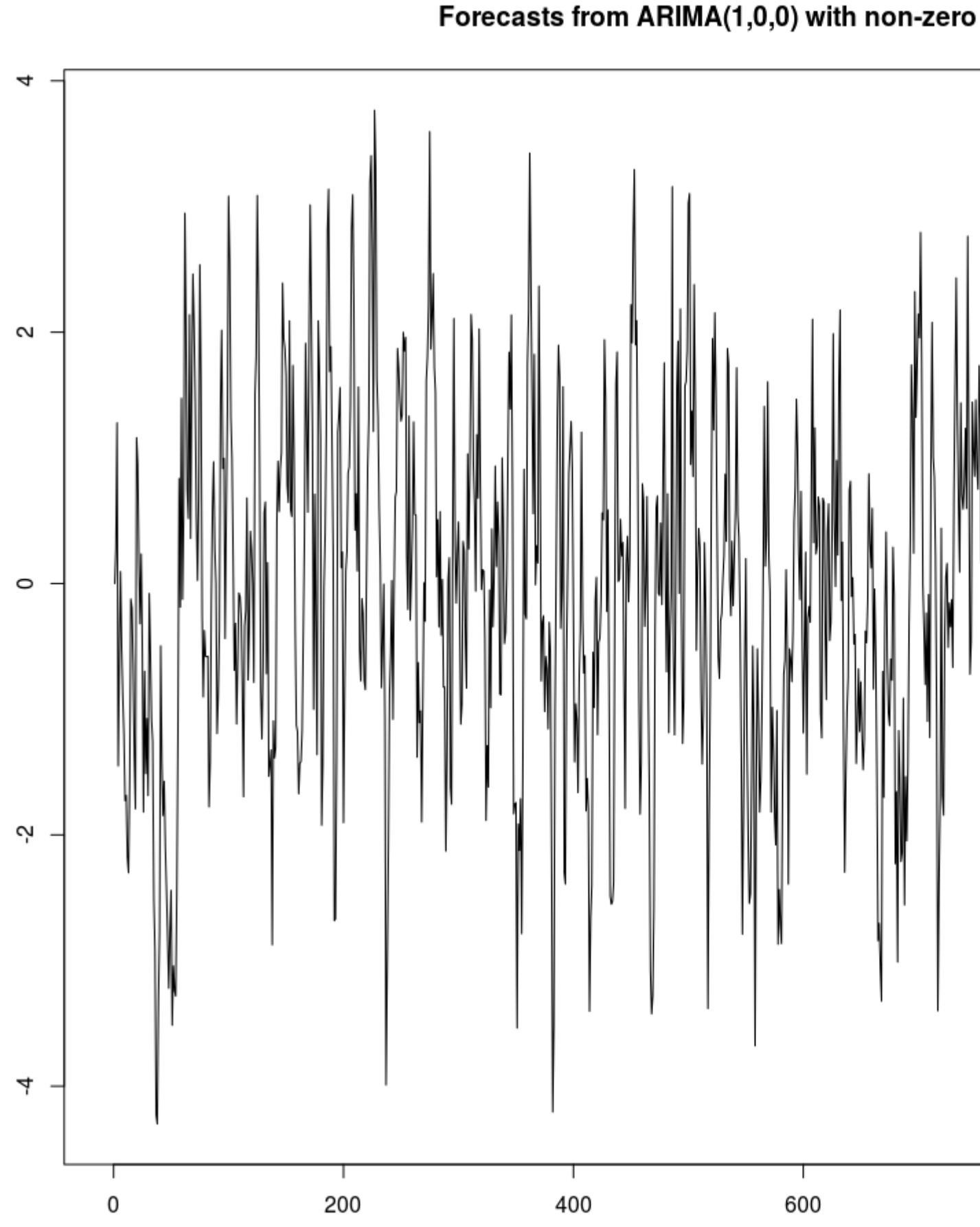

Series fit$resid
ACF
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
Lag

```
#Forecast 10 periods
fcst <- forecast(fit, h = 100)
fcst
     Point Forecast      Lo 80    Hi 80     Lo 95    Hi 95
1001    0.282529070 -0.9940493 1.559107 -1.669829 2.234887
1002    0.173976408 -1.3872262 1.735179 -2.213677 2.561630
1003    0.097554408 -1.5869850 1.782094 -2.478726 2.673835
1004    0.043752667 -1.6986831 1.786188 -2.621073 2.708578
1005    0.005875783 -1.7645535 1.776305 -2.701762 2.713514
...

#Call the point predictions
fcst$mean
# Time Series:
# Start = 1001
# End = 1100
# Frequency = 1
  [1]  0.282529070  0.173976408  0.097554408  0.043752667  0.005875783 -0.020789866 -
0.039562711 -0.052778954
  [9] -0.062083302
...

#Plot the forecast
plot(fcst)
```


www.dbooks.org


Forecasts from ARIMA(1,0,0) with non-zero
4
2
0
-2
-4
0
200
400
600

Read Arima Models online: https://riptutorial.com/r/topic/1725/arima-models

www.dbooks.org

Chapter 8: Arithmetic Operators

Remarks

Nearly all operators in R are really functions. For example, `+` is a function defined as `function (e1, e2) .Primitive("+")` where e1 is the left-hand side of the operator and e2 is the right-hand side of the operator. This means it is possible to accomplish rather counterintuitive effects by masking the `+` in base with a user defined function.

For example:

```
`+` <- function(e1, e2) {e1-e2}

> 3+10
[1] -7
```

Examples

Range and addition

Let's take an example of adding a value to a range (as it could be done in a loop for example):

```
3+1:5
```

Gives:

```
[1] 4 5 6 7 8
```

This is because the range operator `:` has higher precedence than addition operator `+`.

What happens during evaluation is as follows:

- `3+1:5`
- `3+c(1, 2, 3, 4, 5)` expansion of the range operator to make a vector of integers.
- `c(4, 5, 6, 7, 8)` Addition of 3 to each member of the vector.

To avoid this behavior you have to tell the R interpreter how you want it to order the operations with `( )` like this:

```
  (3+1):5
```

Now R will compute what is inside the parentheses before expanding the range and gives:

```
 [1] 4 5
```

Addition and subtraction

The basic math operations are performed mainly on numbers or on vectors (lists of numbers).

1. Using single numbers

We can simple enter the numbers concatenated with `+` for *adding* and `-` for *subtracting*:

```
> 3 + 4.5
# [1] 7.5
> 3 + 4.5 + 2
# [1] 9.5
> 3 + 4.5 + 2 - 3.8
# [1] 5.7
> 3 + NA
#[1] NA
> NA + NA
#[1] NA
> NA - NA
#[1] NA
> NaN - NA
#[1] NaN
> NaN + NA
#[1] NaN
```

We can assign the numbers to *variables* (constants in this case) and do the same operations:

```
> a <- 3; B <- 4.5; cc <- 2; Dd <- 3.8 ;na<-NA;nan<-NaN
> a + B
# [1] 7.5
> a + B + cc
# [1] 9.5
> a + B + cc - Dd
# [1] 5.7
> B-nan
#[1] NaN
> a+na-na
#[1] NA
> a + na
#[1] NA
> B-nan
#[1] NaN
> a+na-na
#[1] NA
```

2. Using vectors

In this case we create vectors of numbers and do the operations using those vectors, or combinations with single numbers. In this case the operation is done considering each element of the vector:

```
> A <- c(3, 4.5, 2, -3.8);
> A
# [1]  3.0  4.5  2.0 -3.8
> A + 2 # Adding a number
```


www.dbooks.org

```
# [1]  5.0  6.5  4.0 -1.8
> 8 - A # number less vector
# [1]  5.0  3.5  6.0 11.8
> n <- length(A) #number of elements of vector A
> n
# [1] 4
> A[-n] + A[n] # Add the last element to the same vector without the last element
# [1] -0.8  0.7 -1.8
> A[1:2] + 3 # vector with the first two elements plus a number
# [1] 6.0 7.5
> A[1:2] - A[3:4] # vector with the first two elements less the vector with elements 3 and 4
# [1] 1.0 8.3
```

We can also use the function `sum` to add all elements of a vector:

```
> sum(A)
# [1] 5.7
> sum(-A)
# [1] -5.7
> sum(A[-n]) + A[n]
# [1] 5.7
```

We must take care with *recycling*, which is one of the characteristics of `R`, a behavior that happens when doing math operations where the length of vectors is different. *Shorter vectors in the expression are recycled as often as need be (perhaps fractionally) until they match the length of the longest vector. In particular a constant is simply repeated*. In this case a Warning is show.

```
> B <- c(3, 5, -3, 2.7, 1.8)
> B
# [1]  3.0  5.0 -3.0  2.7  1.8
> A
# [1]  3.0  4.5  2.0 -3.8
> A + B # the first element of A is repeated
# [1]  6.0  9.5 -1.0 -1.1  4.8
Warning message:
In A + B : longer object length is not a multiple of shorter object length
> B - A # the first element of A is repeated
# [1]  0.0  0.5 -5.0  6.5 -1.2
Warning message:
In B - A : longer object length is not a multiple of shorter object length
```

In this case the correct procedure will be to consider only the elements of the shorter vector:

```
> B[1:n] + A
# [1]  6.0  9.5 -1.0 -1.1
> B[1:n] - A
# [1]  0.0  0.5 -5.0  6.5
```

When using the `sum` function, again all the elements inside the function are added.

```
> sum(A, B)
# [1] 15.2
> sum(A, -B)
# [1] -3.8
> sum(A)+sum(B)
```

```
# [1] 15.2
> sum(A)-sum(B)
# [1] -3.8
```

Read Arithmetic Operators online: https://riptutorial.com/r/topic/4389/arithmetic-operators

www.dbooks.org

Chapter 9: Bar Chart

Introduction

The purpose of the bar plot is to display the frequencies (or proportions) of levels of a factor variable. For example, a bar plot is used to pictorially display the frequencies (or proportions) of individuals in various socio-economic (factor) groups(levels-high, middle, low). Such a plot will help to provide a visual comparison among the various factor levels.

Examples

barplot() function

In barplot, factor-levels are placed on the x-axis and frequencies (or proportions) of various factor-levels are considered on the y-axis. For each factor-level one bar of uniform width with heights being proportional to factor level frequency (or proportion) is constructed.

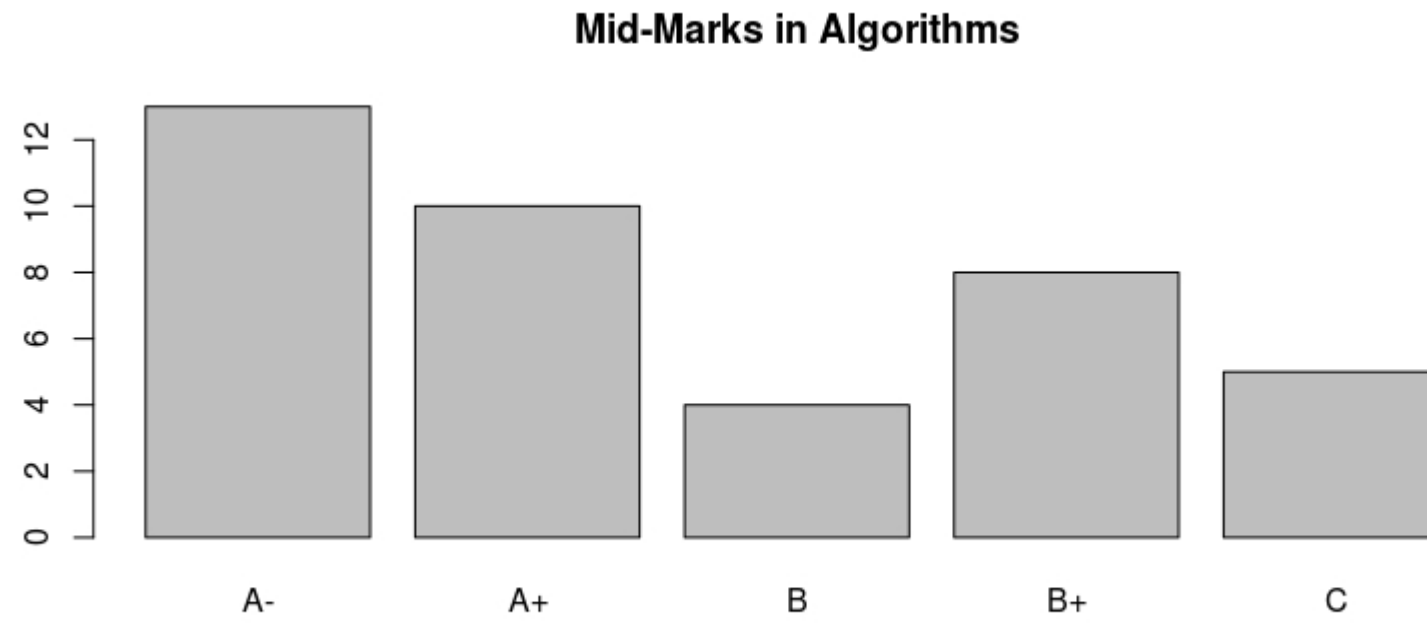
The `barplot()` function is in the graphics package of the R's System Library. The `barplot()` function must be supplied at least one argument. The R help calls this as `heights`, which must be either vector or a matrix. If it is vector, its members are the various factor-levels.

To illustrate `barplot()`, consider the following data preparation:

```
> grades<-c("A+","A-","B+","B","C")
> Marks<-sample(grades,40,replace=T,prob=c(.2,.3,.25,.15,.1))
> Marks
[1] "A+" "A-" "B+" "A-" "A+" "B"  "A+" "B+" "A-" "B"  "A+" "A-"
[13] "A-" "B+" "A-" "A-" "A-" "A-" "A+" "A-" "A+" "A+" "C"  "C"
[25] "B"  "C"  "B+" "C"  "B+" "B+" "B+" "A+" "B+" "A-" "A+" "A-"
[37] "A-" "B"  "C"  "A+"
>
```

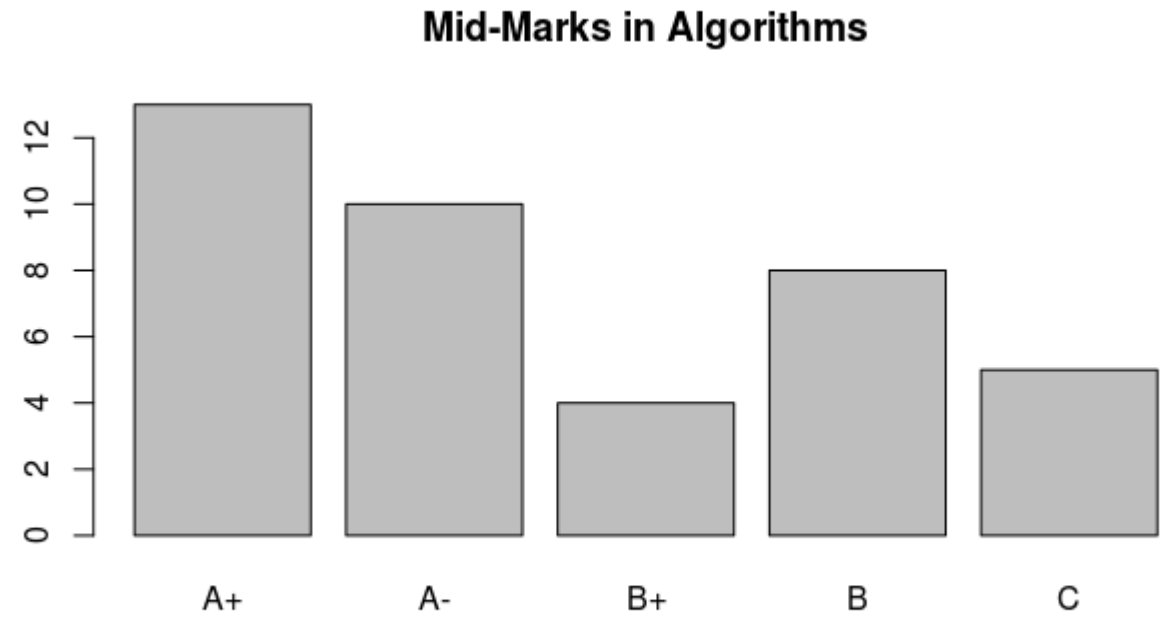
A bar chart of the Marks vector is obtained from

```
> barplot(table(Marks),main="Mid-Marks in Algorithms")
```



Notice that, the barplot() function places the factor levels on the x-axis in the `lexicographical order` of the levels. Using the parameter `names.arg`, the bars in plot can be placed in the order as stated in the vector, *grades*.

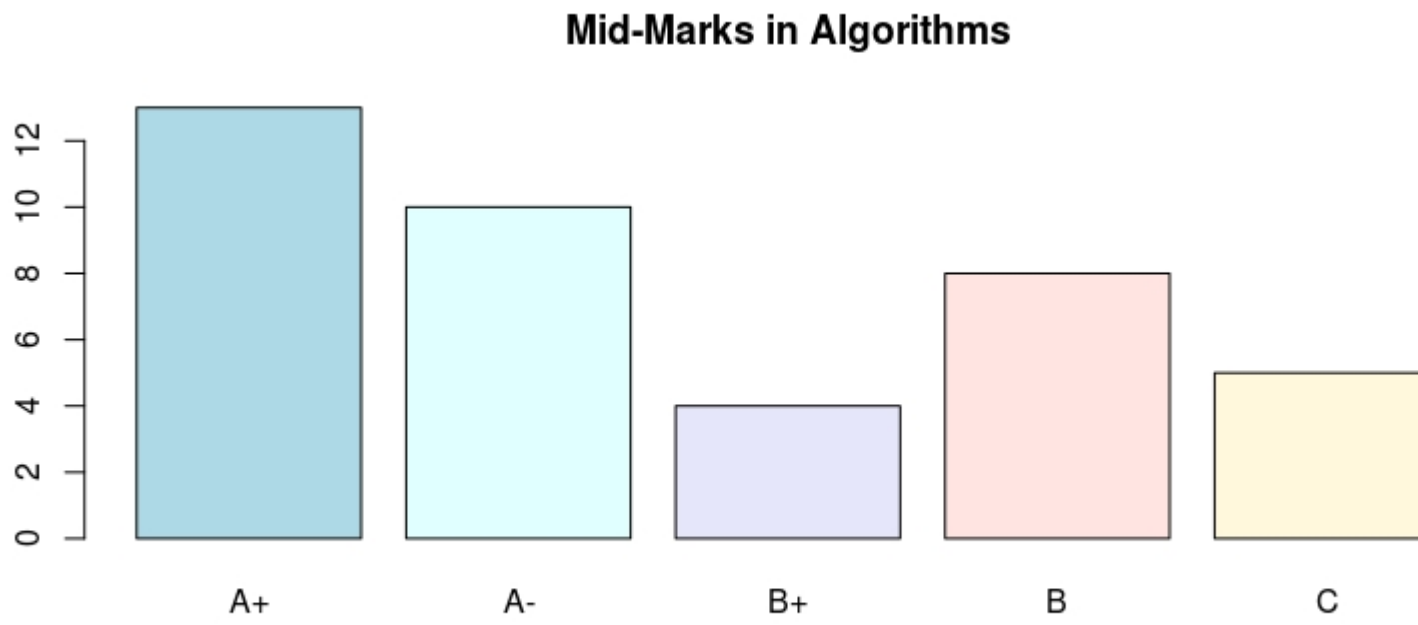
```
# plot to the desired horizontal axis labels
> barplot(table(Marks),names.arg=grades ,main="Mid-Marks in Algorithms")
```



Colored bars can be drawn using the `col=` parameter.

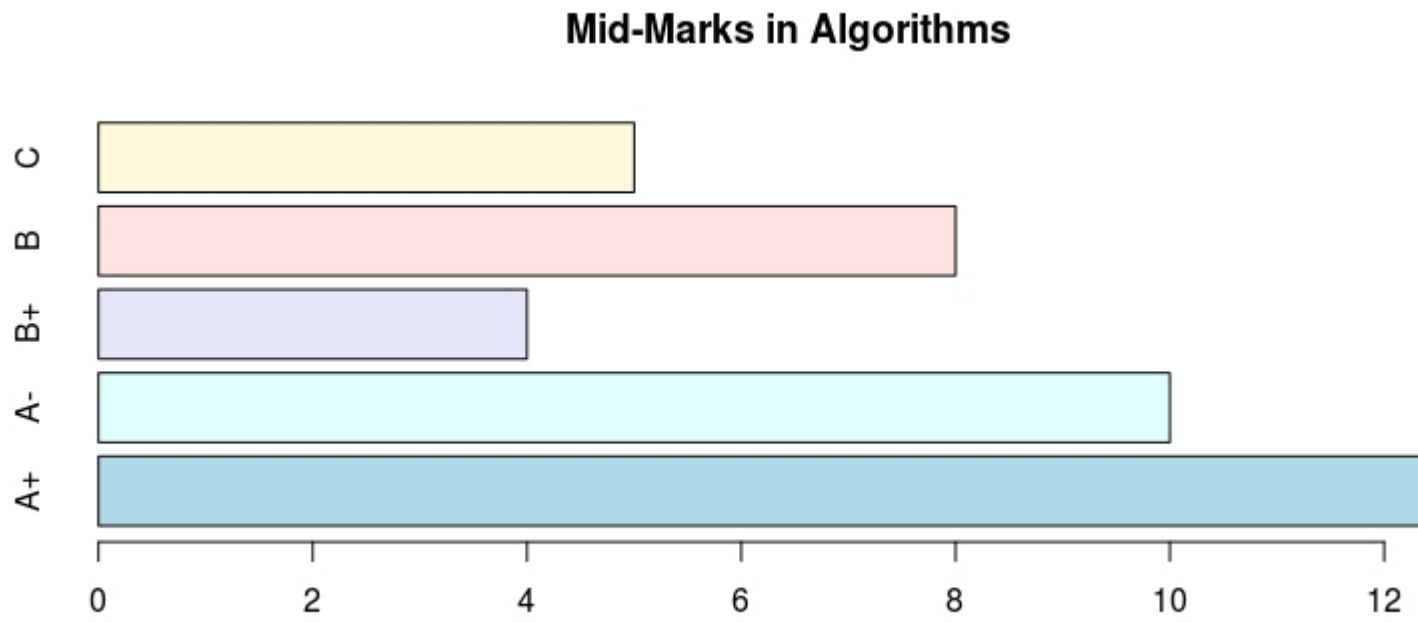
www.dbooks.org

```
> barplot(table(Marks),names.arg=grades,col = c("lightblue",
        "lightcyan", "lavender", "mistyrose",  "cornsilk"),
         main="Mid-Marks in Algorithms")
```



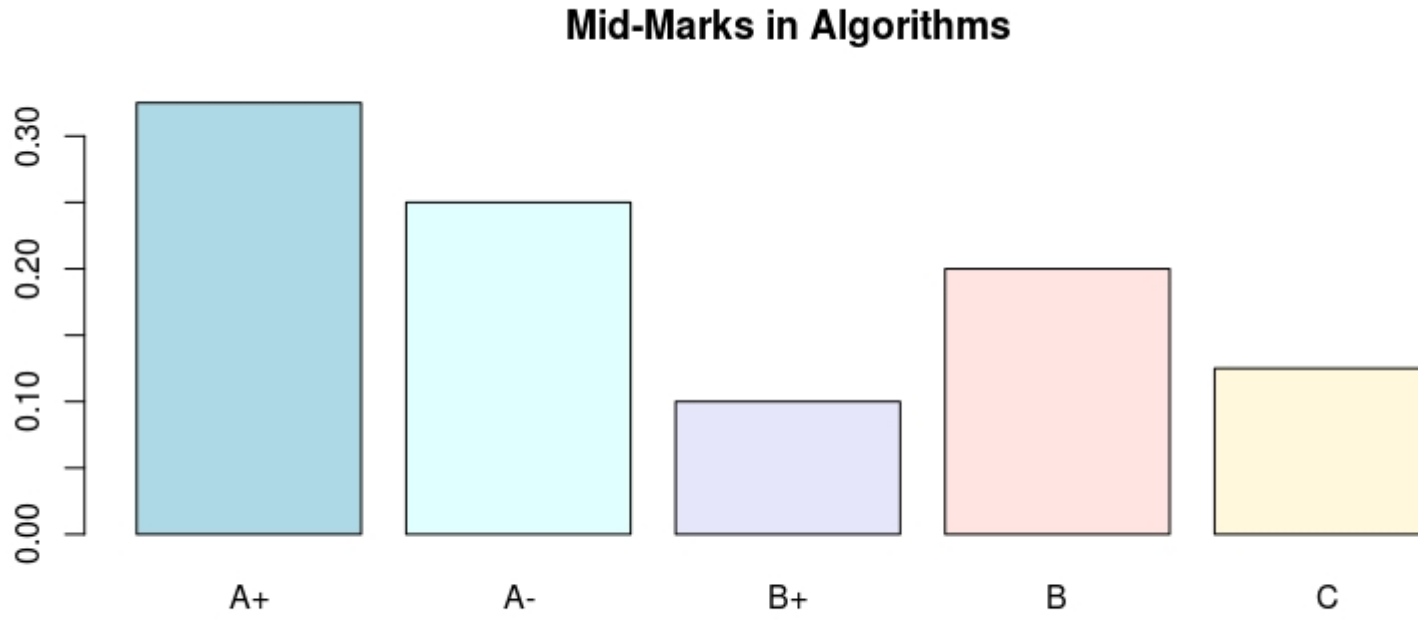
A bar chart with *horizontal bars* can be obtained as follows:

```
> barplot(table(Marks),names.arg=grades,horiz=TRUE,col = c("lightblue",
          "lightcyan", "lavender", "mistyrose",  "cornsilk"),
           main="Mid-Marks in Algorithms")
```



A bar chart with *proportions* on the y-axis can be obtained as follows:

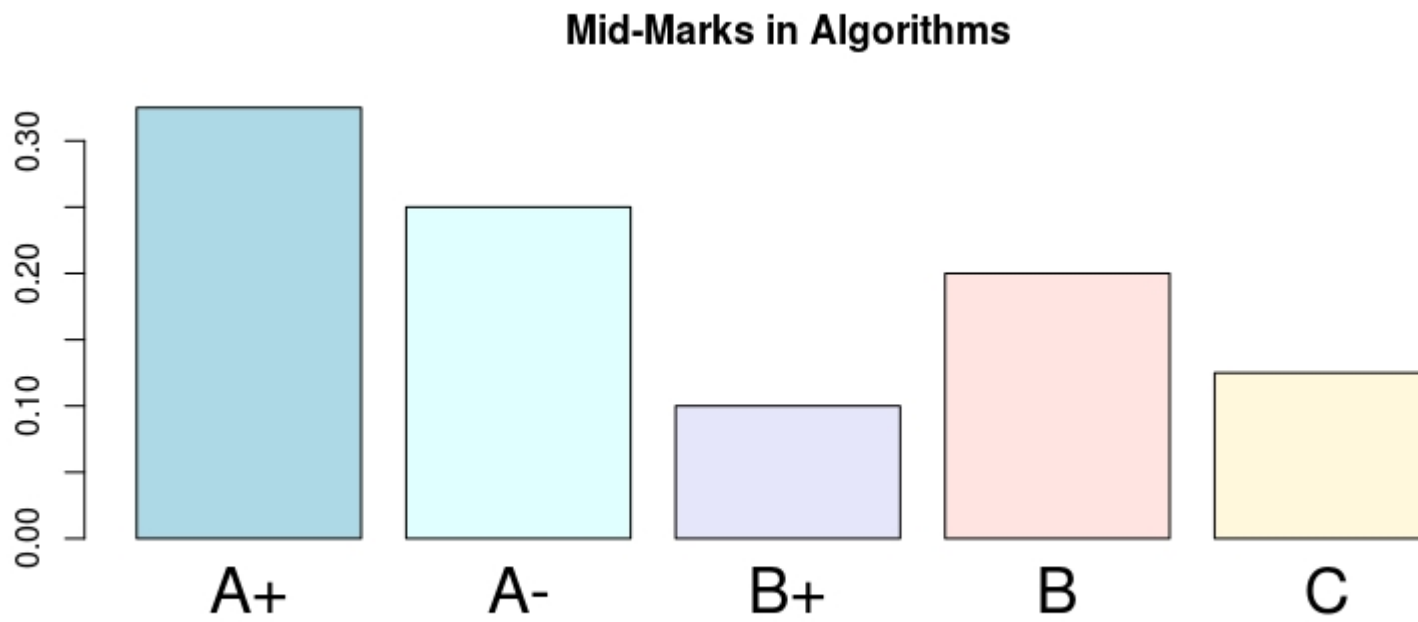
```
> barplot(prop.table(table(Marks)),names.arg=grades,col = c("lightblue",
           "lightcyan", "lavender", "mistyrose",  "cornsilk"),
            main="Mid-Marks in Algorithms")
```



The sizes of the factor-level names on the x-axis can be increased using `cex.names` parameter.

www.dbooks.org

```
> barplot(prop.table(table(Marks)),names.arg=grades,col = c("lightblue",
          "lightcyan", "lavender", "mistyrose",  "cornsilk"),
           main="Mid-Marks in Algorithms",cex.names=2)
```

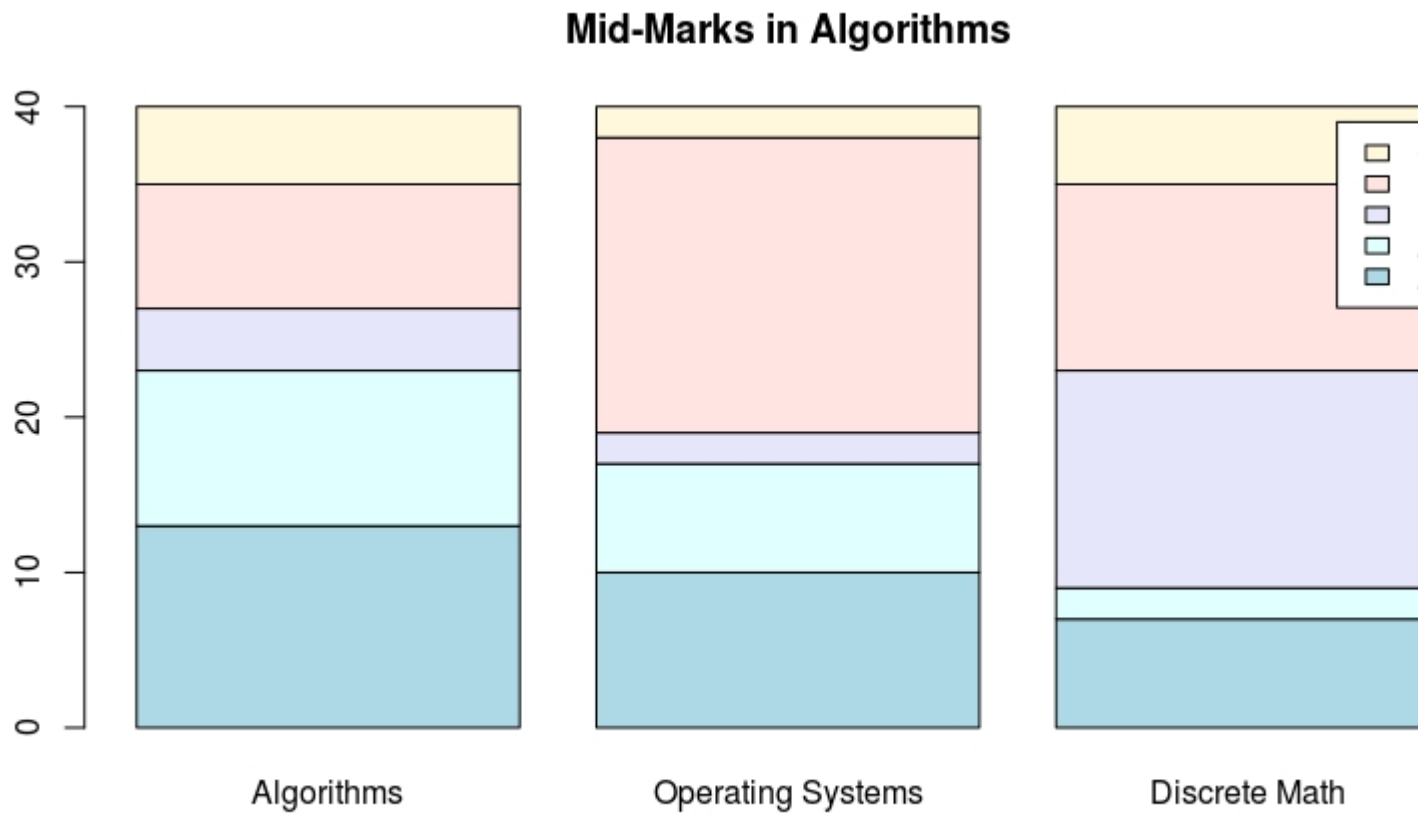


The `heights` parameter of the `barplot()` could be a matrix. For example it could be matrix, where the columns are the various subjects taken in a course, the rows could be the labels of the grades. Consider the following matrix:

```
> gradTab
     Algorithms Operating Systems Discrete Math
  A-         13                10             7
  A+         10                 7             2
  B           4                 2            14
  B+          8                19            12
  C           5                 2             5
```

To draw a stacked bar, simply use the command:

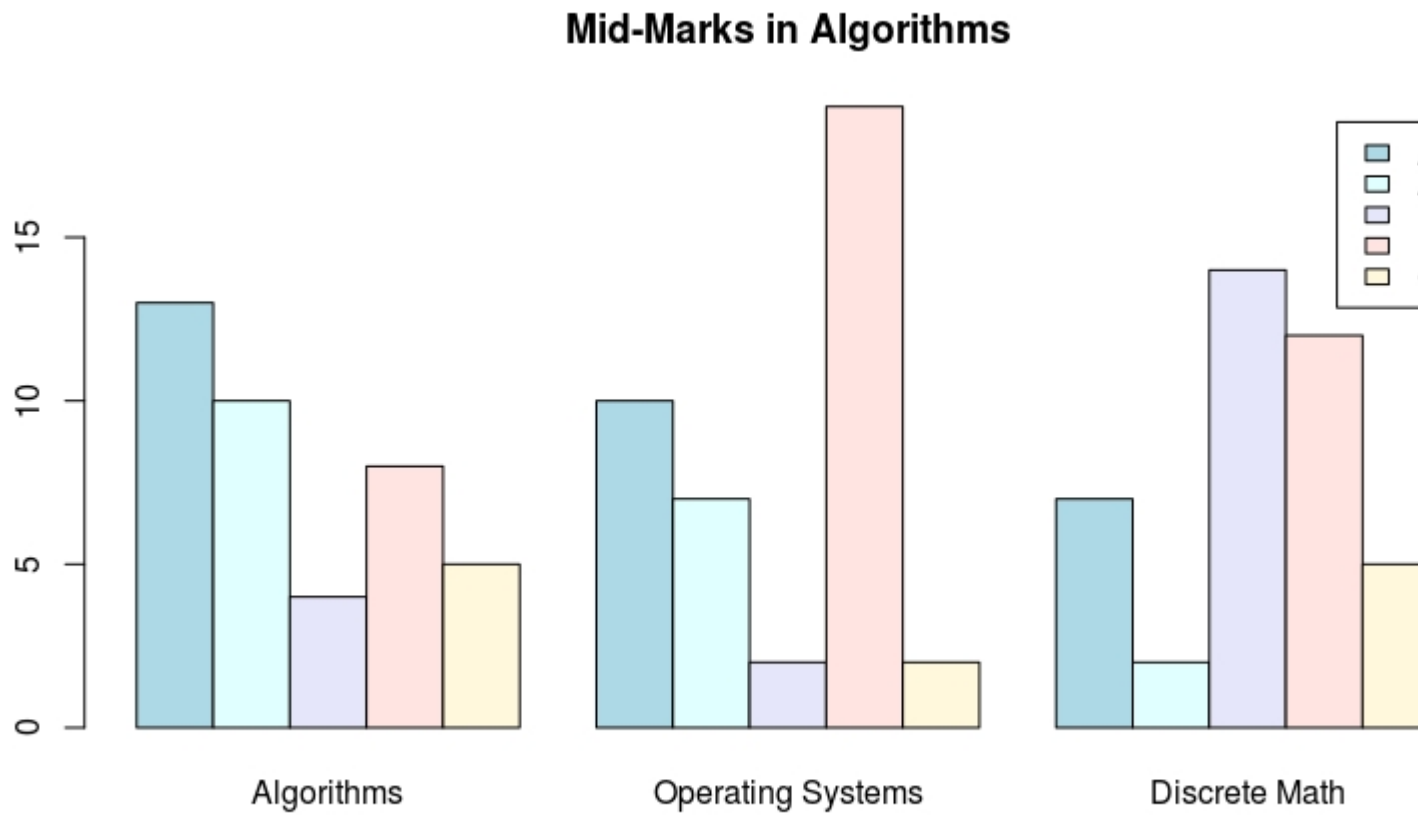
```
> barplot(gradTab,col = c("lightblue","lightcyan",
        "lavender", "mistyrose",  "cornsilk"),legend.text = grades,
         main="Mid-Marks in Algorithms")
```



To draw a juxtaposed bars, use the `besides` parameter, as given under:

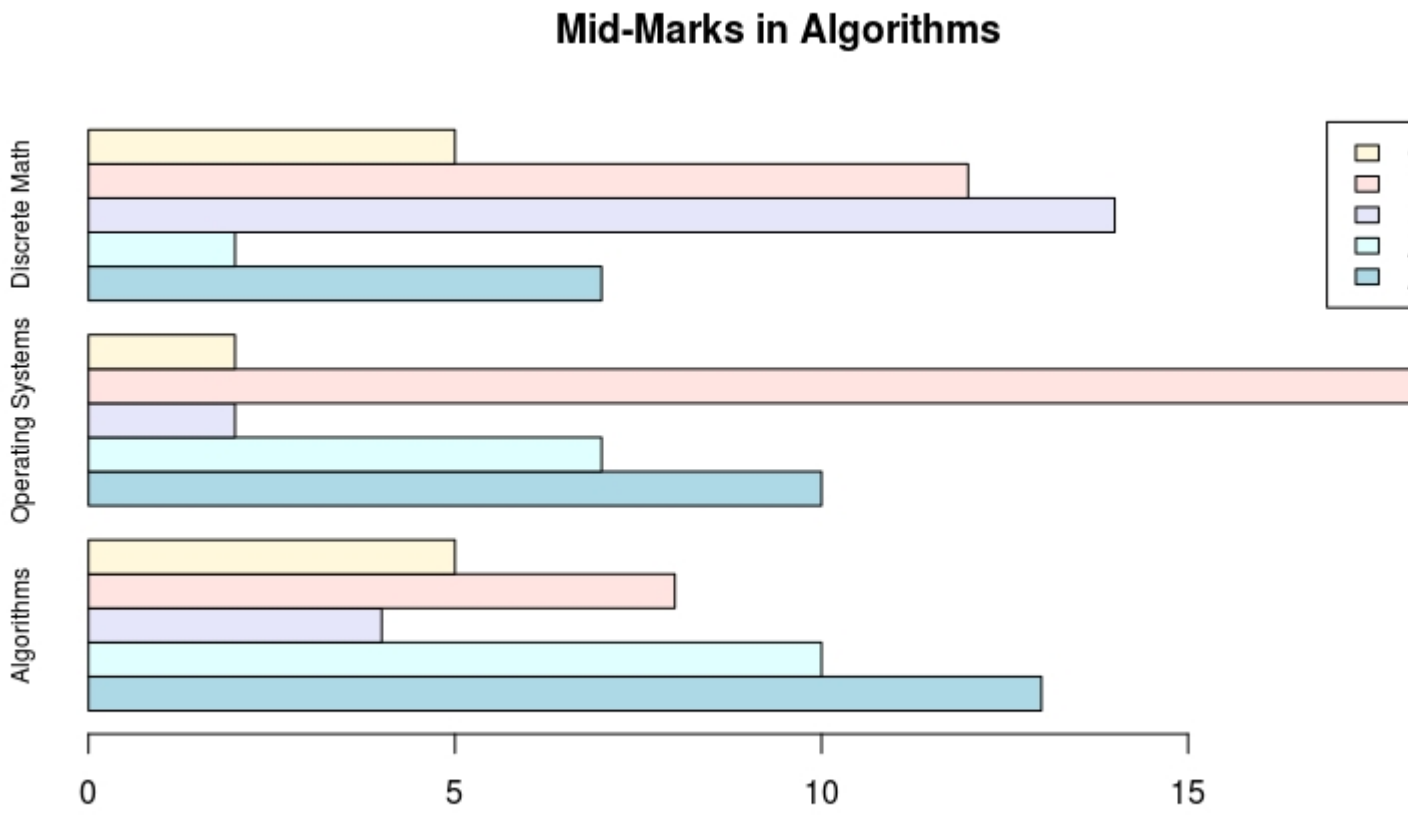
```
> barplot(gradTab,beside = T,col = c("lightblue","lightcyan",
      "lavender", "mistyrose",  "cornsilk"),legend.text = grades,
       main="Mid-Marks in Algorithms")
```


www.dbooks.org



A horizontal bar chart can be obtained using `horiz=T` parameter:

```
> barplot(gradTab,beside = T,horiz=T,col = c("lightblue","lightcyan",
       "lavender", "mistyrose",  "cornsilk"),legend.text = grades,
        cex.names=.75,main="Mid-Marks in Algorithms")
```



Read Bar Chart online: https://riptutorial.com/r/topic/8091/bar-chart

www.dbooks.org

Chapter 10: Base Plotting

Parameters

Parameter	Details
`x`	x-axis variable. May supply either `data$variablex` or `data[,x]`
`y`	y-axis variable. May supply either `data$variabley` or `data[,y]`
`main`	Main title of plot
`sub`	Optional subtitle of plot
`xlab`	Label for x-axis
`ylab`	Label for y-axis
`pch`	Integer or character indicating plotting symbol
`col`	Integer or string indicating color
`type`	Type of plot. `"p"` for points, `"l"` for lines, `"b"` for both, `"c"` for the lines part alone of `"b"`, `"o"` for both 'overplotted', `"h"` for 'histogram'-like (or 'high-density') vertical lines, `"s"` for stair steps, `"S"` for other steps, `"n"` for no plotting

Remarks

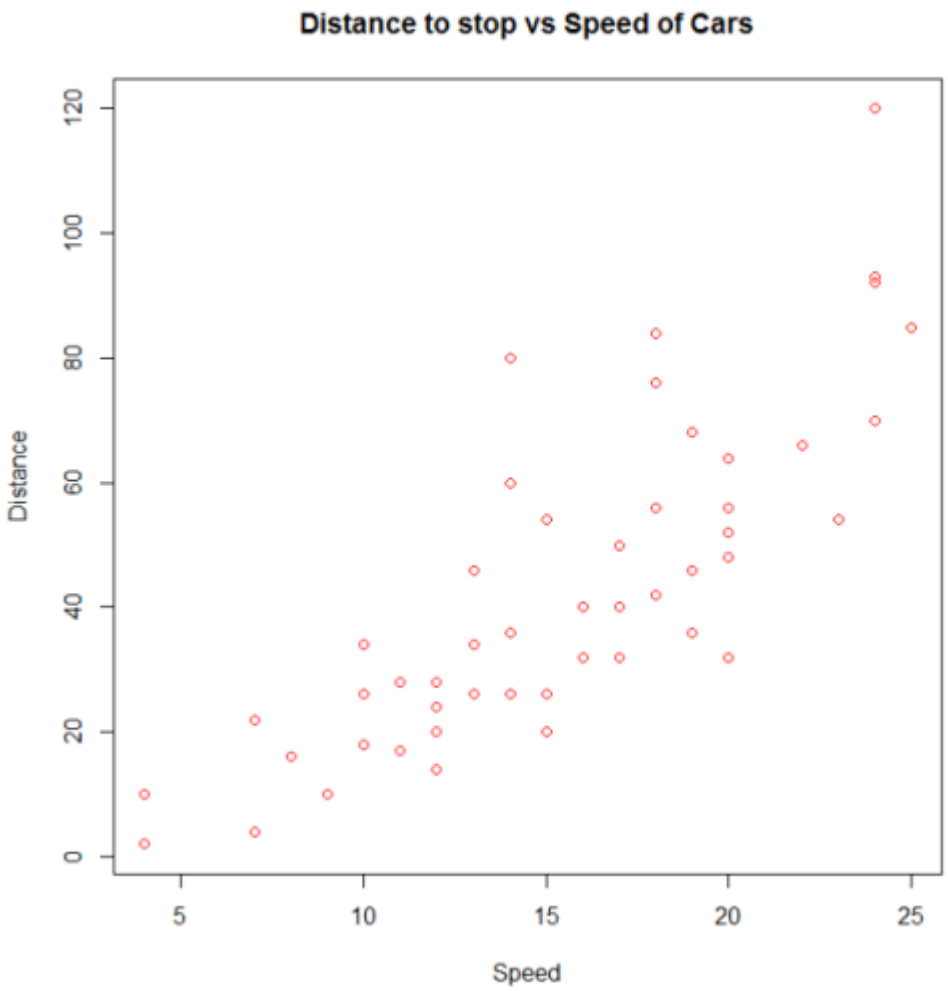
The items listed in the "Parameters" section is a small fraction of hte possible parameters that can be modified or set by the `par` function. See `par` for a more complete list. In addition all the graphics devices, including the system specific interactive graphics devices will have a set of parameters that can customize the output.

Examples

Basic Plot

A basic plot is created by calling `plot()`. Here we use the built-in `cars` data frame that contains the speed of cars and the distances taken to stop in the 1920s. (To find out more about the dataset, use help(cars)).

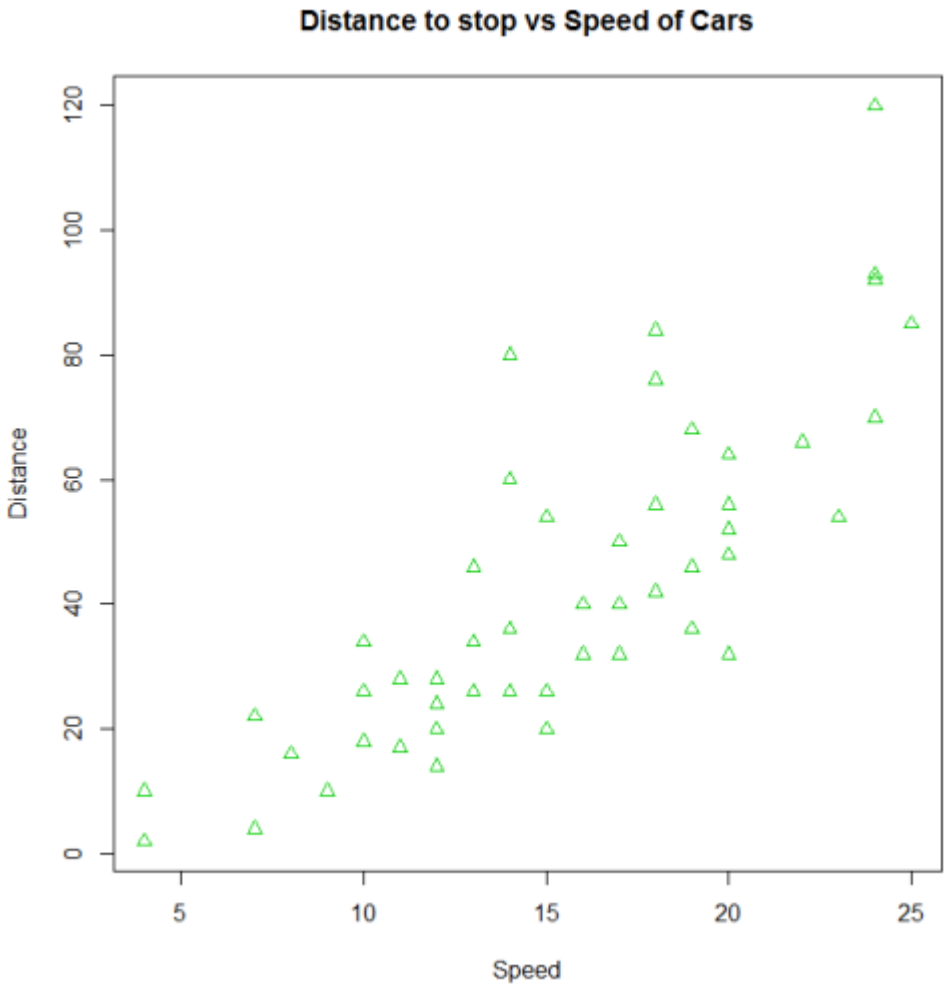
```
plot(x = cars$speed, y = cars$dist, pch = 1, col = 1,
     main = "Distance vs Speed of Cars",
     xlab = "Speed", ylab = "Distance")
```



We can use many other variations in the code to get the same result. We can also change the parameters to obtain different results.

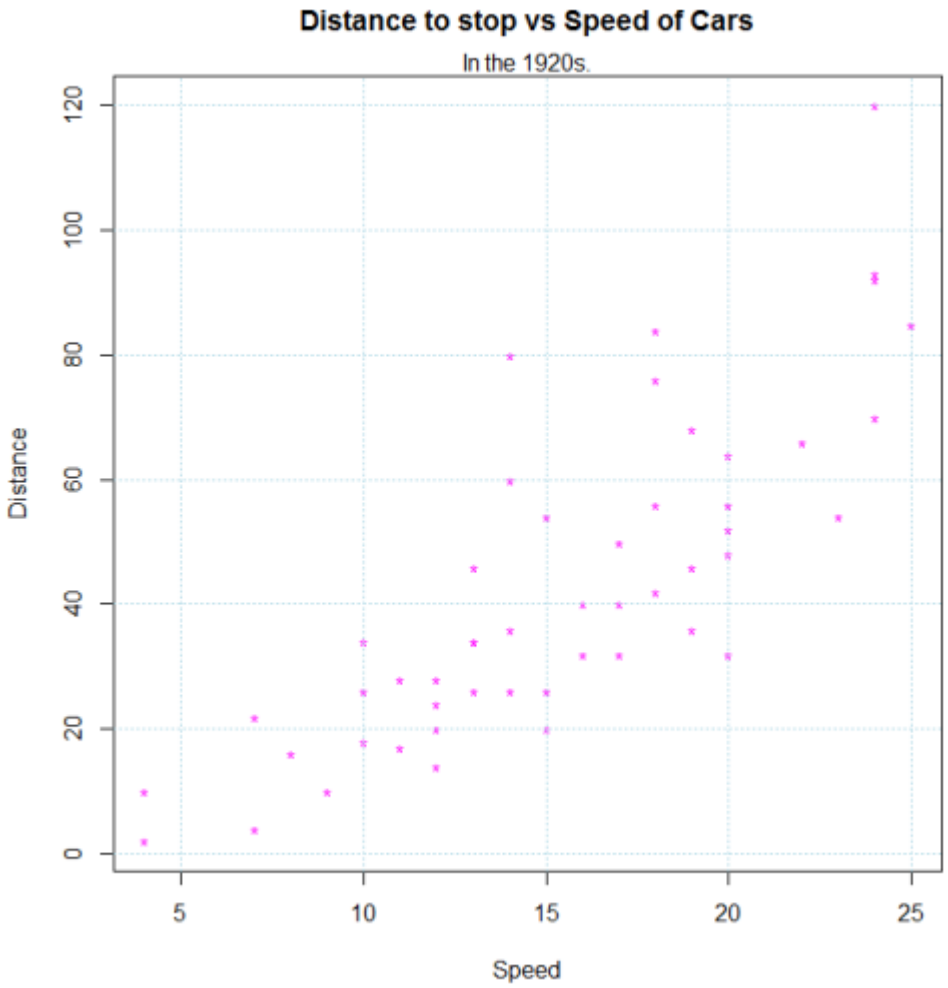
```
with(cars, plot(dist~speed, pch = 2, col = 3,
     main = "Distance to stop vs Speed of Cars",
     xlab = "Speed", ylab = "Distance"))
```


www.dbooks.org



Additional features can be added to this plot by calling `points()`, `text()`, `mtext()`, `lines()`, `grid()`, etc.

```
plot(dist~speed, pch = "*", col = "magenta", data=cars,
     main = "Distance to stop vs Speed of Cars",
     xlab = "Speed", ylab = "Distance")
mtext("In the 1920s.")
grid(,col="lightblue")
```



Matplot

`matplot` is useful for quickly plotting multiple sets of observations from the same object, particularly from a matrix, on the same graph.

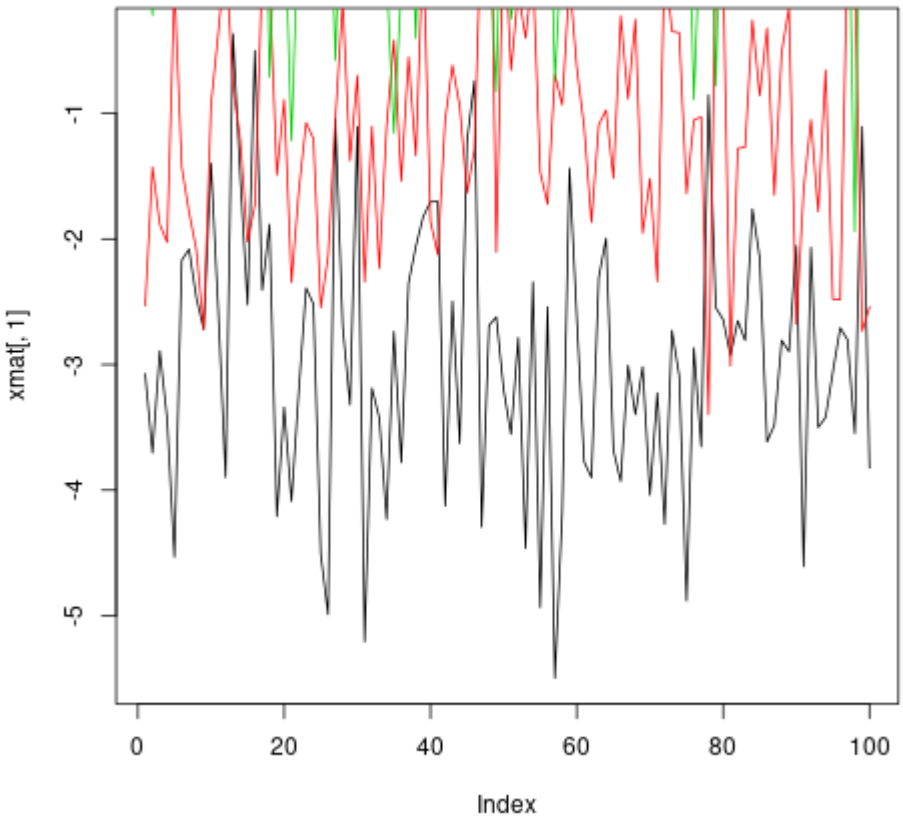
Here is an example of a matrix containing four sets of random draws, each with a different mean.

```
xmat <- cbind(rnorm(100, -3), rnorm(100, -1), rnorm(100, 1), rnorm(100, 3))
head(xmat)
#           [,1]        [,2]       [,3]     [,4]
# [1,] -3.072793 -2.53111494  0.6168063 3.780465
# [2,] -3.702545 -1.42789347 -0.2197196 2.478416
# [3,] -2.890698 -1.88476126  1.9586467 5.268474
# [4,] -3.431133 -2.02626870  1.1153643 3.170689
# [5,] -4.532925  0.02164187  0.9783948 3.162121
# [6,] -2.169391 -1.42699116  0.3214854 4.480305
```

One way to plot all of these observations on the same graph is to do one `plot` call followed by three more `points` or `lines` calls.

```
plot(xmat[,1], type = 'l')
lines(xmat[,2], col = 'red')
lines(xmat[,3], col = 'green')
lines(xmat[,4], col = 'blue')
```

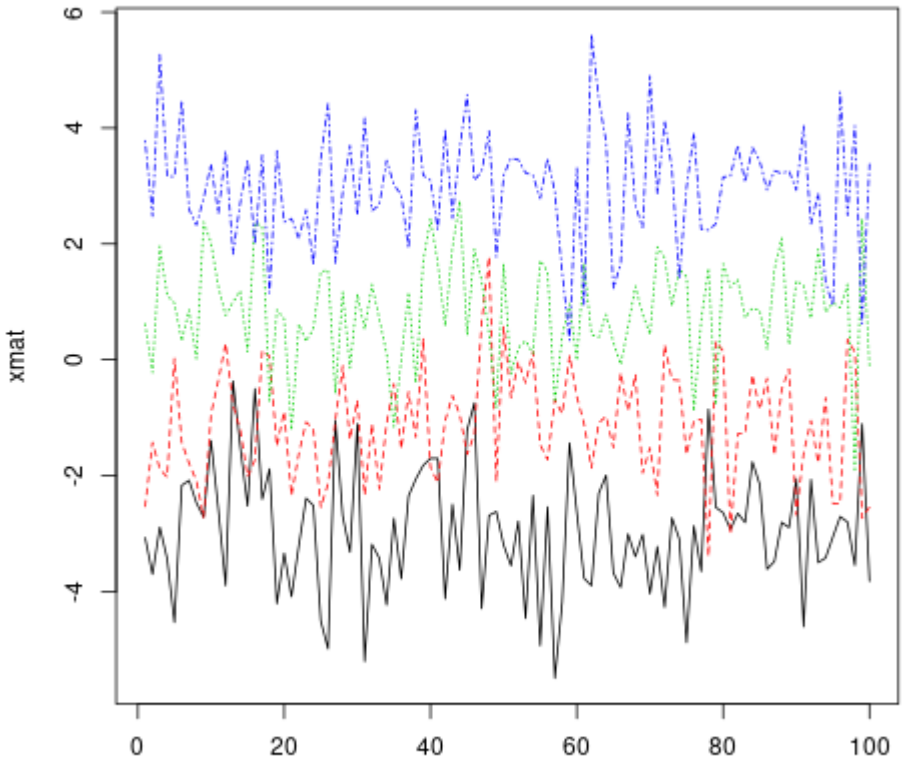

www.dbooks.org



However, this is both tedious, and causes problems because, among other things, by default the axis limits are fixed by `plot` to fit only the first column.

Much more convenient in this situation is to use the `matplot` function, which only requires one call and automatically takes care of axis limits *and* changing the aesthetics for each column to make them distinguishable.

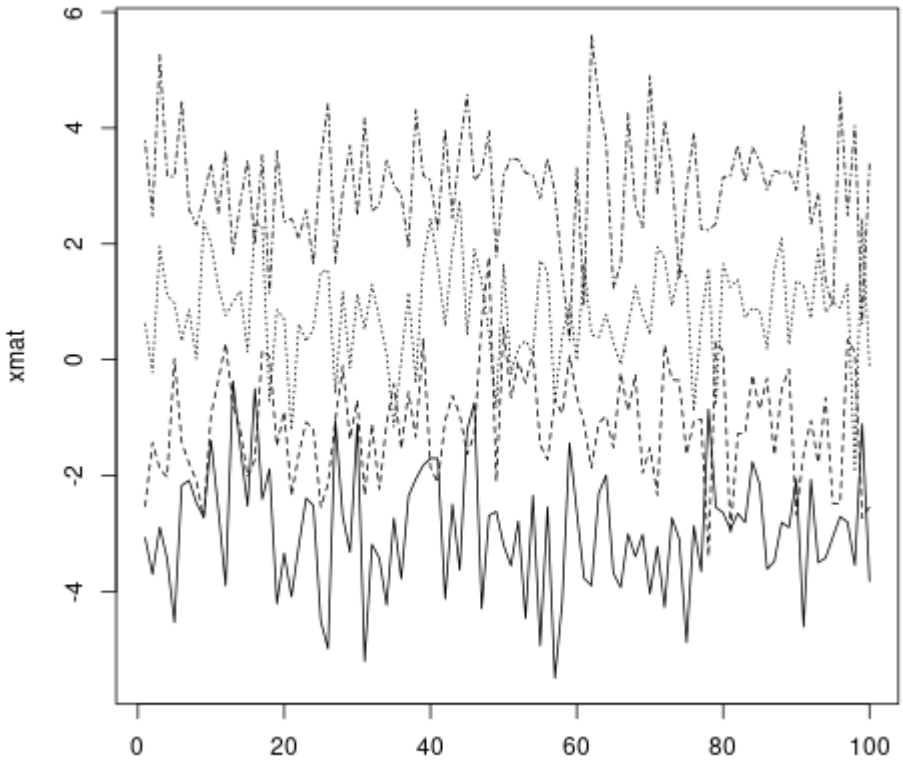
```
matplot(xmat, type = 'l')
```



Note that, by default, `matplot` varies both color (`col`) and linetype (`lty`) because this increases the number of possible combinations before they get repeated. However, any (or both) of these aesthetics can be fixed to a single value...

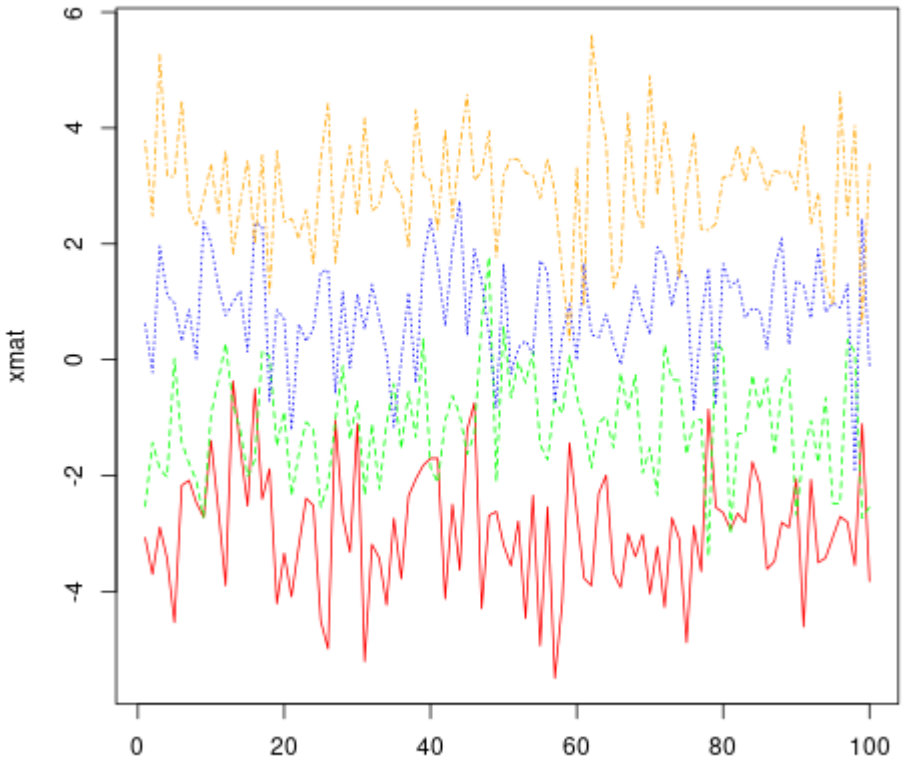
```
matplot(xmat, type = 'l', col = 'black')
```


www.dbooks.org



...or a custom vector (which will recycle to the number of columns, following standard R vector recycling rules).

```
matplot(xmat, type = 'l', col = c('red', 'green', 'blue', 'orange'))
```

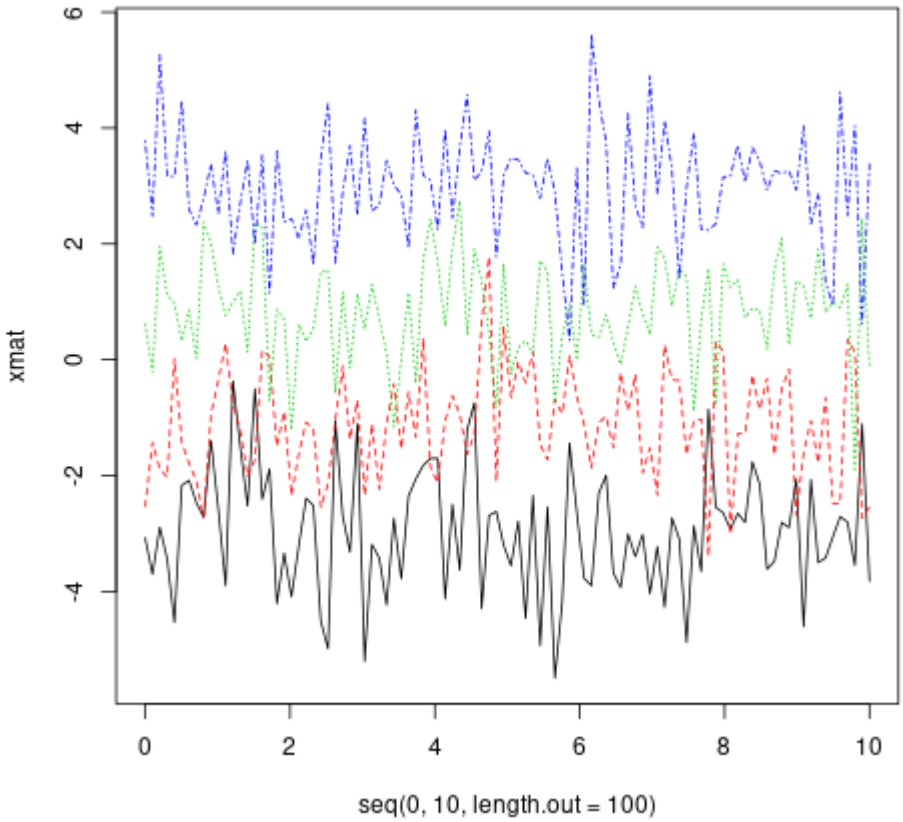


Standard graphical parameters, including `main`, `xlab`, `xmin`, work exactly the same way as for `plot`. For more on those, see `?par`.

Like `plot`, if given only one object, `matplot` assumes it's the `y` variable and uses the indices for `x`. However, `x` and `y` can be specified explicitly.

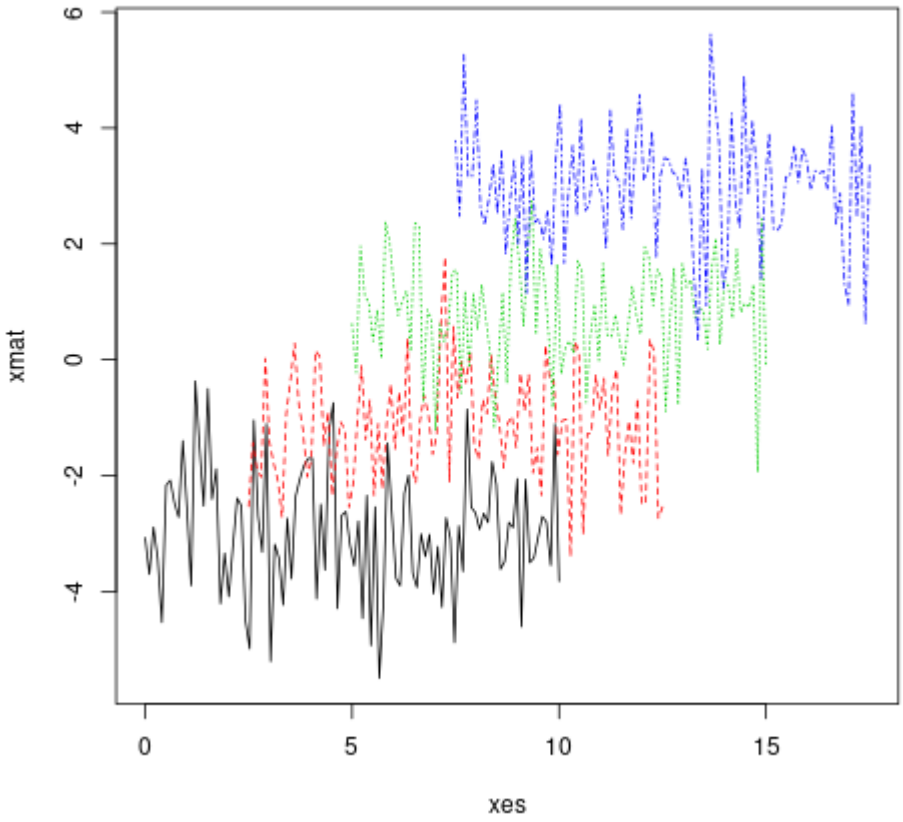
```
matplot(x = seq(0, 10, length.out = 100), y = xmat, type='l')
```


www.dbooks.org



In fact, both `x` and `y` can be matrices.

```
xes <- cbind(seq(0, 10, length.out = 100),
             seq(2.5, 12.5, length.out = 100),
             seq(5, 15, length.out = 100),
             seq(7.5, 17.5, length.out = 100))
matplot(x = xes, y = xmat, type = 'l')
```

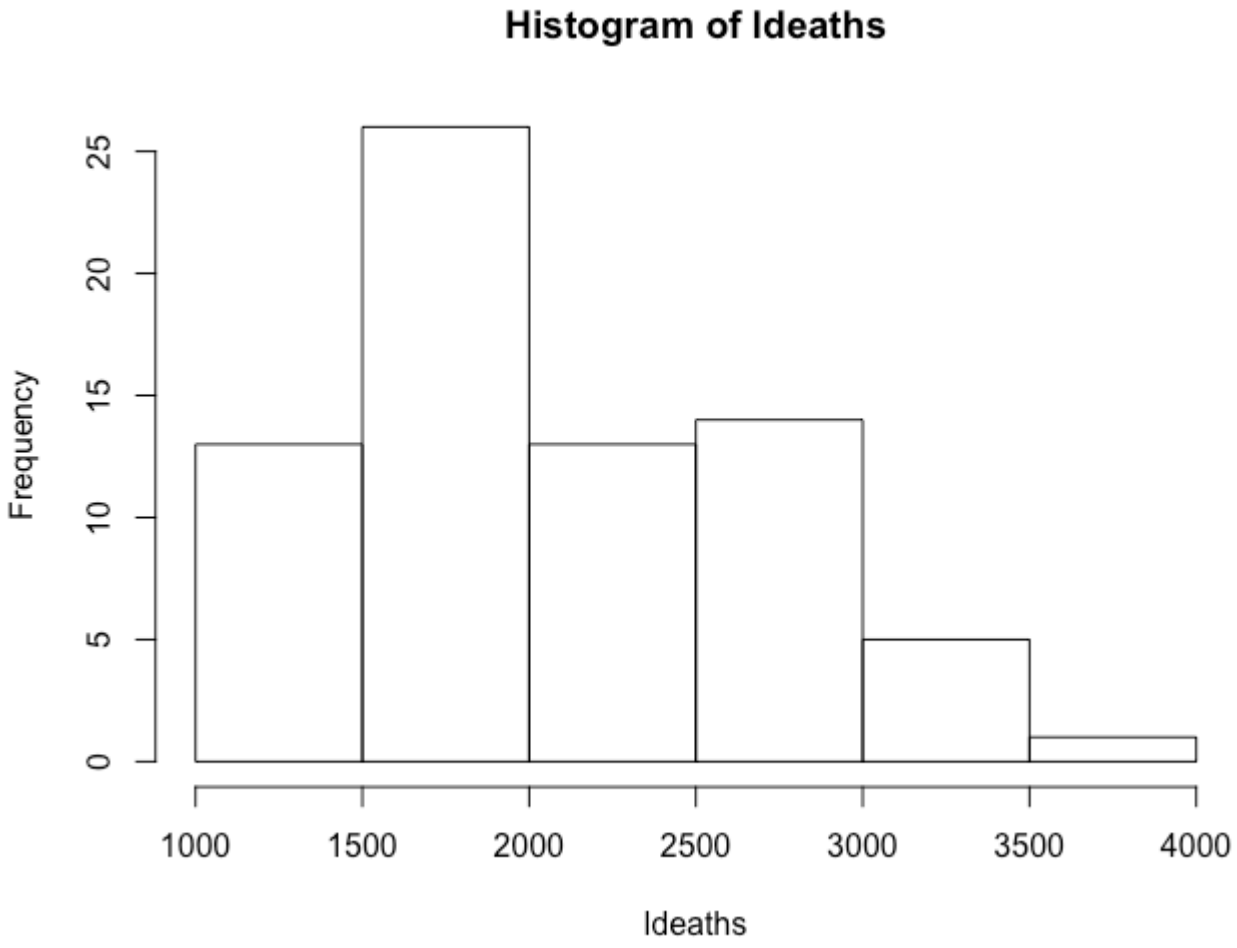


Histograms

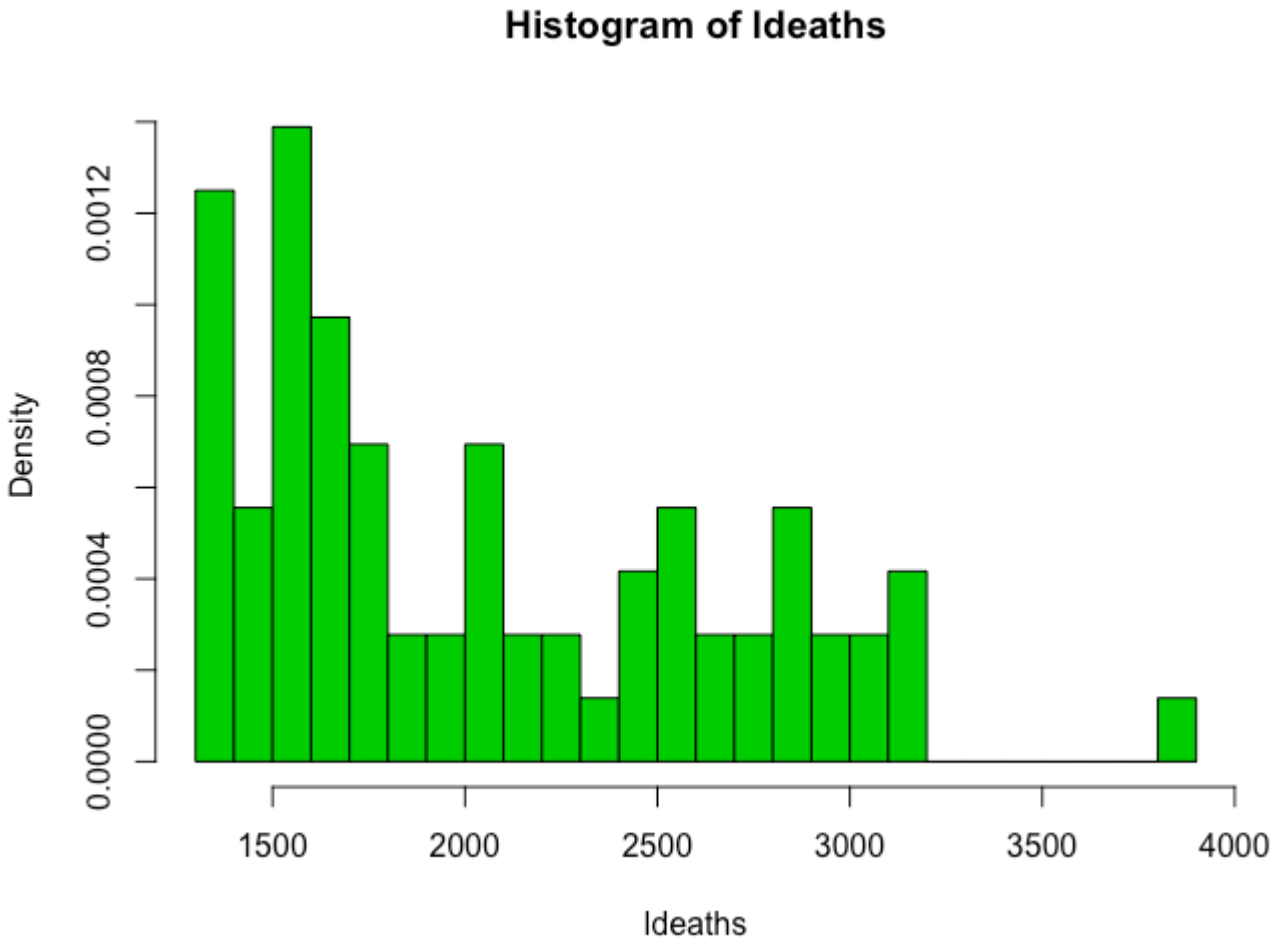
Histograms allow for a pseudo-plot of the underlying distribution of the data.

```
hist(ldeaths)
```


www.dbooks.org




```
hist(ldeaths, breaks = 20, freq = F, col = 3)
```



Combining Plots

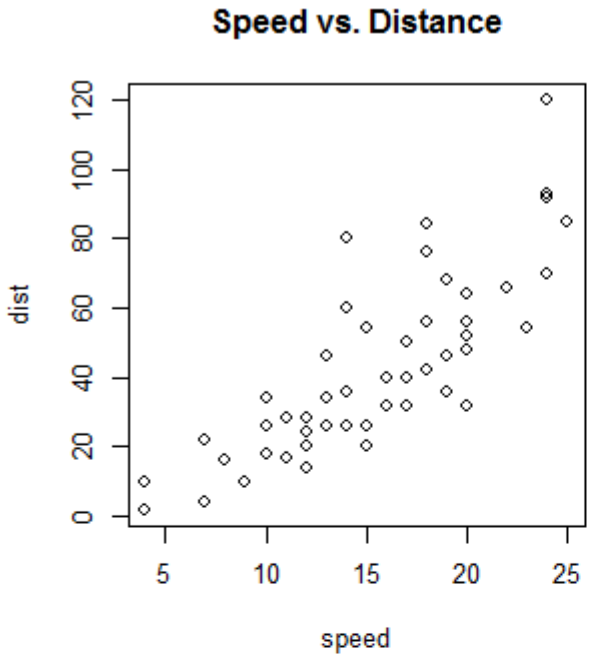
It's often useful to combine multiple plot types in one graph (for example a Barplot next to a Scatterplot.) R makes this easy with the help of the functions `par()` and `layout()`.

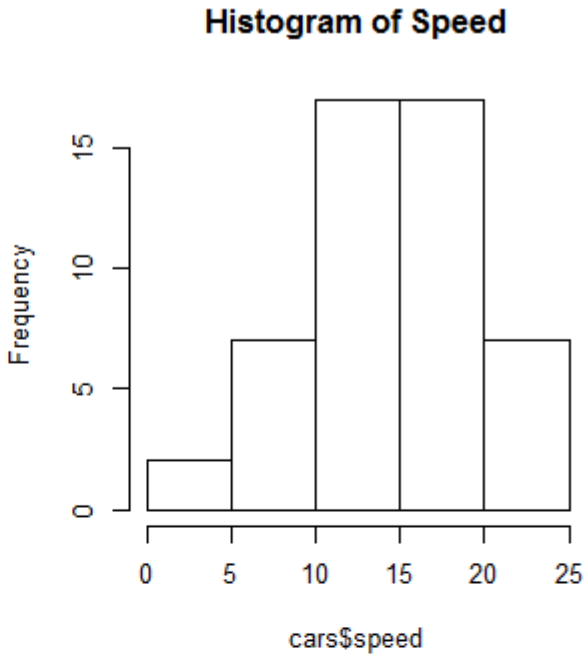
`par()`

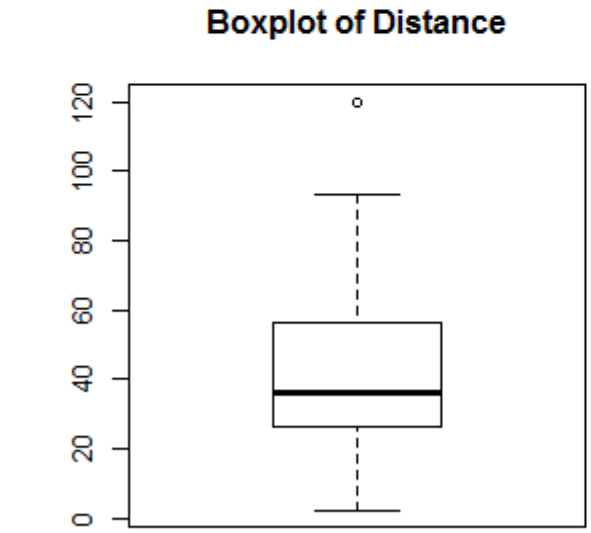
`par` uses the arguments `mfrow` or `mfcol` to create a matrix of nrows and ncols `c(nrows, ncols)` which will serve as a grid for your plots. The following example shows how to combine four plots in one graph:

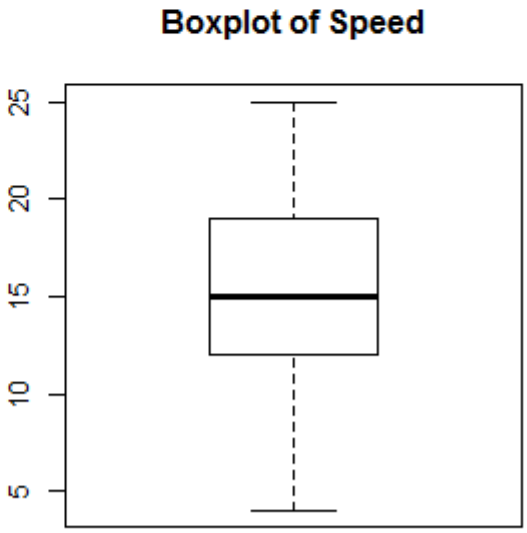
```
par(mfrow=c(2,2))
plot(cars, main="Speed vs. Distance")
hist(cars$speed, main="Histogram of Speed")
boxplot(cars$dist, main="Boxplot of Distance")
boxplot(cars$speed, main="Boxplot of Speed")
```


www.dbooks.org





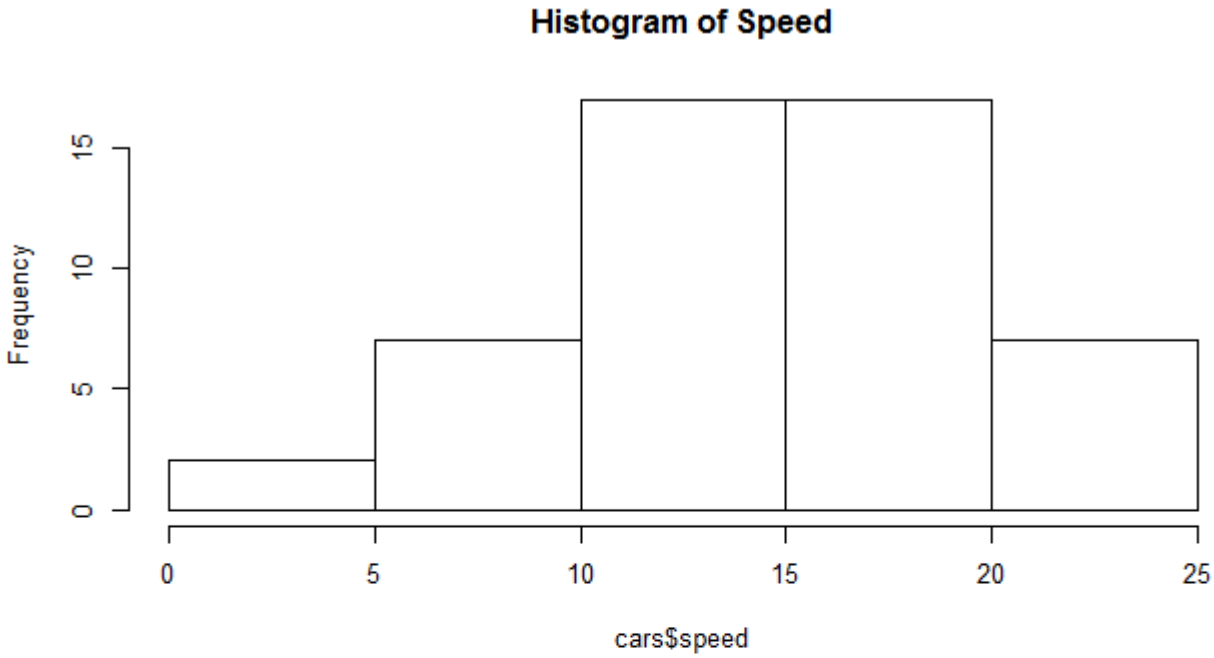


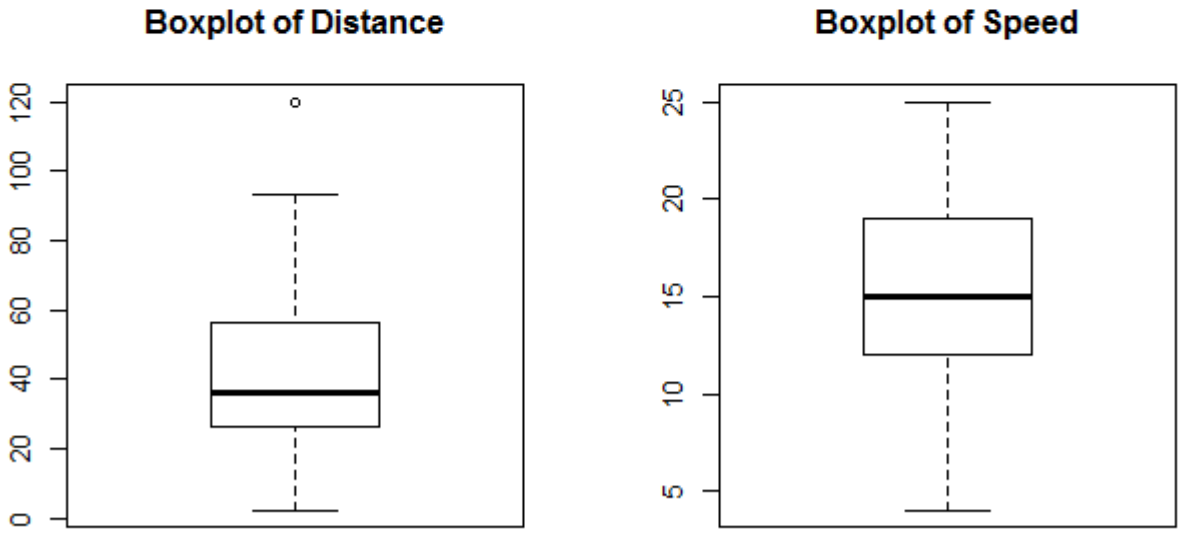


`layout()`

The `layout()` is more flexible and allows you to specify the location and the extent of each plot within the final combined graph. This function expects a matrix object as an input:

```
layout(matrix(c(1,1,2,3), 2,2, byrow=T))
hist(cars$speed, main="Histogram of Speed")
boxplot(cars$dist, main="Boxplot of Distance")
boxplot(cars$speed, main="Boxplot of Speed")
```





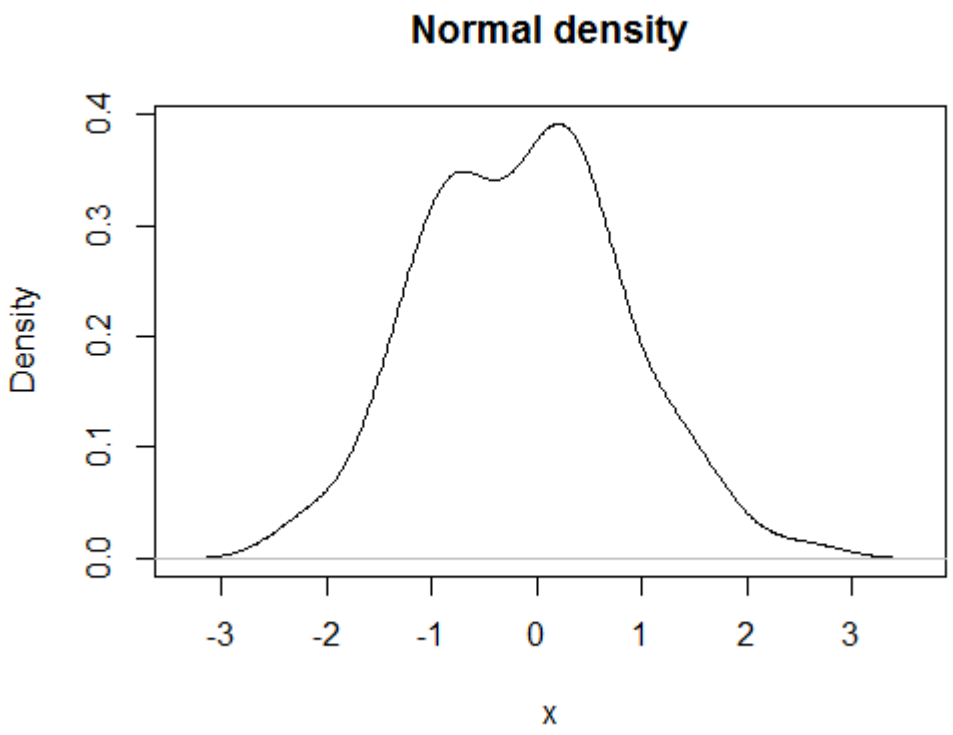
Density plot

A very useful and logical follow-up to histograms would be to plot the smoothed density function of a random variable. A basic plot produced by the command

```
plot(density(rnorm(100)),main="Normal density",xlab="x")
```

would look like

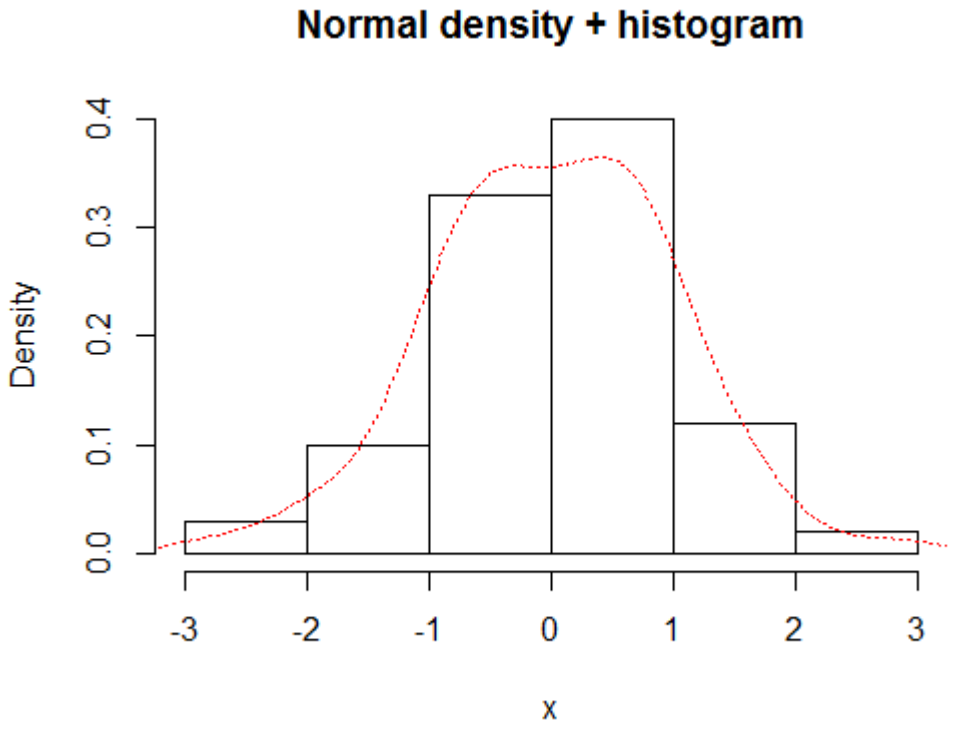
www.dbooks.org



You can overlay a histogram and a density curve with

```
x=rnorm(100)
hist(x,prob=TRUE,main="Normal density + histogram")
lines(density(x),lty="dotted",col="red")
```

which gives



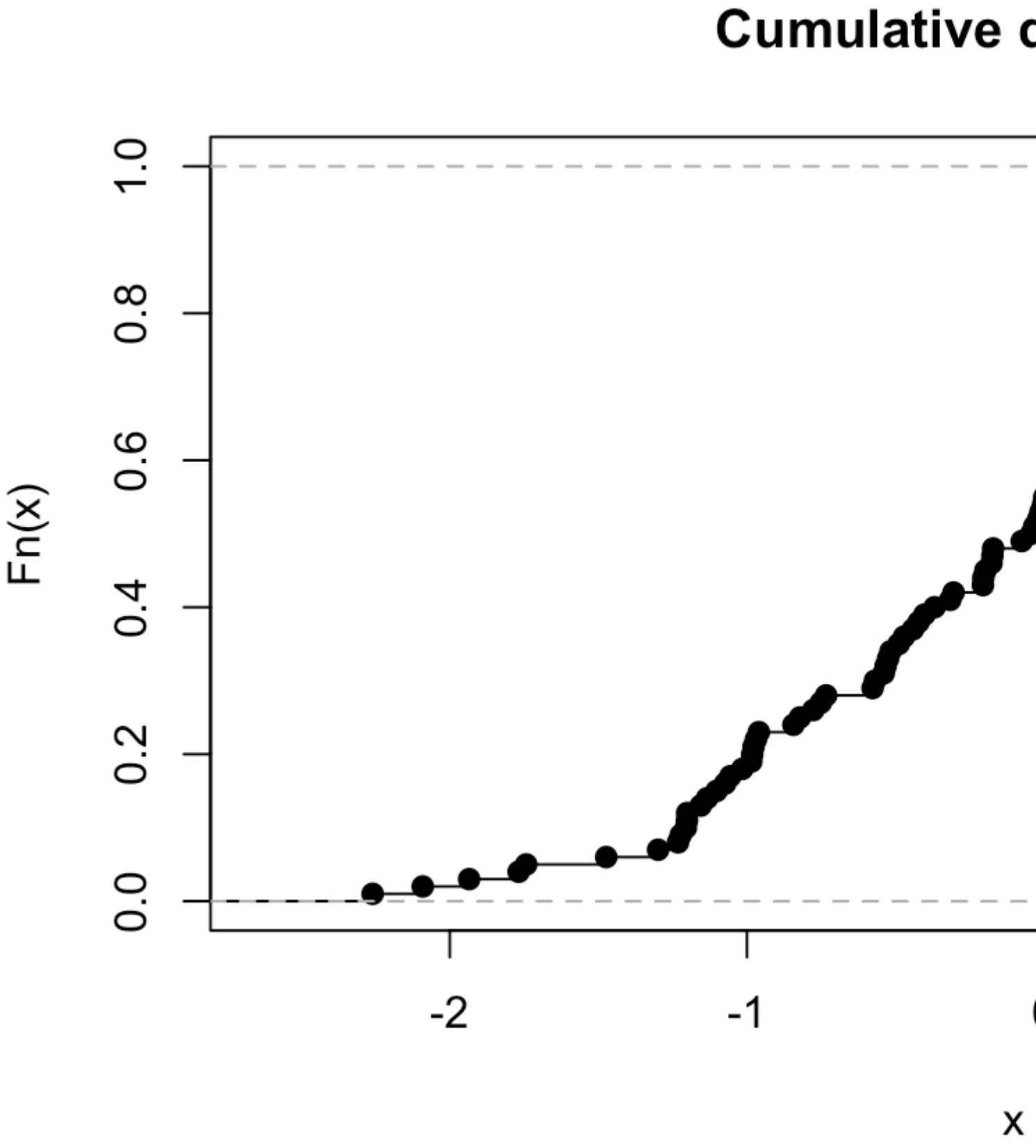
Empirical Cumulative Distribution Function

A very useful and logical follow-up to histograms and density plots would be the Empirical Cumulative Distribution Function. We can use the function `ecdf()` for this purpose. A basic plot produced by the command

```
plot(ecdf(rnorm(100)),main="Cumulative distribution",xlab="x")
```

would look like

www.dbooks.org



Getting Started with R_Plots

- **Scatterplot**

You have two vectors and you want to plot them.

```
x_values <- rnorm(n = 20 , mean = 5 , sd = 8) #20 values generated from Normal(5,8)
y_values <- rbeta(n = 20 , shape1 = 500 , shape2 = 10) #20 values generated from Beta(500,10)
```

If you want to make a plot which has the `y_values` in vertical axis and the `x_values`in horizontal axis, you can use the following commands:

```
plot(x = x_values, y = y_values, type = "p") #standard scatter-plot
plot(x = x_values, y = y_values, type = "l") # plot with lines
plot(x = x_values, y = y_values, type = "n") # empty plot
```

You can type `?plot()` in the console to read about more options.

- **Boxplot**

You have some variables and you want to examine their Distributions

```
#boxplot is an easy way to see if we have some outliers in the data.

z<- rbeta(20 , 500 , 10) #generating values from beta distribution
z[c(19 , 20)] <- c(0.97 , 1.05) # replace the two last values with outliers
boxplot(z) # the two points are the outliers of variable z.
```

- **Histograms**

Easy way to draw histograms

```
hist(x = x_values) # Histogram for x vector
hist(x = x_values, breaks = 3) #use breaks to set the numbers of bars you want
```

- **Pie_charts**

If you want to visualize the frequencies of a variable just draw pie

First we have to generate data with frequencies, for example :

```
P <- c(rep('A' , 3) , rep('B' , 10) , rep('C' , 7) )
t <- table(P) # this is a frequency matrix of variable P
pie(t) # And this is a visual version of the matrix above
```

Read Base Plotting online: https://riptutorial.com/r/topic/1377/base-plotting

www.dbooks.org

Chapter 11: Bibliography in RMD

Parameters

Parameter in YAML header	Detail
`toc`	table of contents
`number_sections`	numbering the sections automatically
`bibliography`	path to the bibliography file
`csl`	path to the style file

Remarks

- The purpose of this documentation is integrating an academic bibliography in a RMD file.
- To use the documentation given above, you have to install `rmarkdown` in R via `install.packages("rmarkdown")`.
- Sometimes Rmarkdown removes the hyperlinks of the citations. The solution for this is adding the following code to your YAML header: `link-citations: true`
- The bibliography may have any of these formats:

Format	File extension
MODS	.mods
BibLaTeX	.bib
BibTeX	.bibtex
RIS	.ris
EndNote	.enl
EndNote XML	.xml
ISI	.wos
MEDLINE	.medline
Copac	.copac
JSON citeproc	.json

Examples

Specifying a bibliography and cite authors

The most important part of your RMD file is the YAML header. For writing an academic paper, I suggest to use PDF output, numbered sections and a table of content (toc).

```
---
title: "Writing an academic paper in R"
author: "Author"
date: "Date"
output:
  pdf_document:
    number_sections: yes
toc: yes
bibliography: bibliography.bib
---
```

In this example, our file `bibliography.bib` looks like this:

```
@ARTICLE{Meyer2000,
  AUTHOR="Bernd Meyer",
  TITLE="A constraint-based framework for diagrammatic reasoning",
  JOURNAL="Applied Artificial Intelligence",
  VOLUME= "14",
  ISSUE = "4",
  PAGES= "327--344",
  YEAR=2000
}
```

To cite an author mentioned in your .bib file write `@` and the bibkey, e.g. `Meyer2000`.

```
# Introduction

`@Meyer2000` results in @Meyer2000.

`@Meyer2000 [p. 328]` results in @Meyer2000 [p. 328]

`[@Meyer2000]` results in [@Meyer2000]

`[-@Meyer2000]` results in [-@Meyer2000]

# Summary

# References
```

Rendering the RMD file via RStudio (Ctrl+Shift+K) or via console `rmarkdown::render("<path-to-your-RMD-file">)` results in the following output:

www.dbooks.org

Writing an academic paper in

Author

Date

Contents

1 Introduction

2 Summary

References

1 Introduction

`@Meyer2000` results in Meyer (2000).

`@Meyer2000 [p. 328]` results in Meyer (2000, 328)

`[@Meyer2000]` results in (Meyer 2000)

`[-@Meyer2000]` results in (2000)

2 Summary

References

Meyer, Bernd. 2000. "A Constraint-Based Framework for Diagrammatic Rea
Intelligence 14 (4): 327–44.

will use a Chicago author-date format for citations and references. To use another style, you will need to specify a CSL 1.0 style file in the csl metadata field. In the following a often used citation style, the elsevier style, is presented (download at https://github.com/citation-style-language/styles). The style-file has to be stored in the same directory as the RMD file OR the absolute path to the file has to be submitted.

To use another style then the default one, the following code is used:

```
---
title: "Writing an academic paper in R"
author: "Author"
date: "Date"
output:
  pdf_document:
    number_sections: yes
toc: yes
bibliography: bibliography.bib
csl: elsevier-harvard.csl
---

# Introduction

`@Meyer2000` results in @Meyer2000.

`@Meyer2000 [p. 328]` results in @Meyer2000 [p. 328]

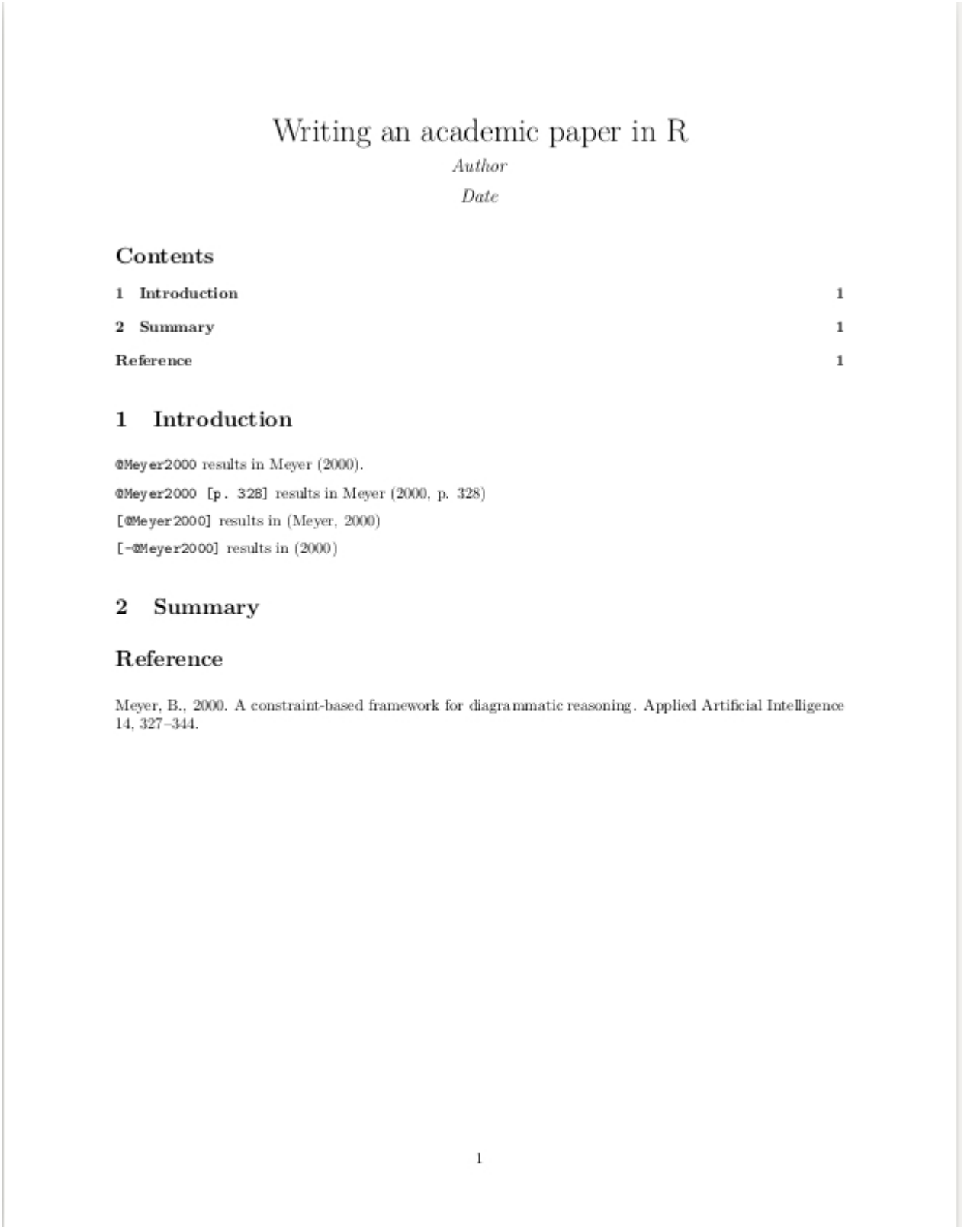
`[@Meyer2000]` results in [@Meyer2000]

`[-@Meyer2000]` results in [-@Meyer2000]

# Summary

# Reference
```


www.dbooks.org



Writing an academic paper in R

Author

Date

Contents

1 Introduction 1

2 Summary 1

Reference 1

1 Introduction

`@Meyer2000` results in Meyer (2000).

`@Meyer2000 [p. 328]` results in Meyer (2000, p. 328)

`[@Meyer2000]` results in (Meyer, 2000)

`[-@Meyer2000]` results in (2000)

2 Summary

Reference

Meyer, B., 2000. A constraint-based framework for diagrammatic reasoning. Applied Artificial Intelligence 14, 327–344.

1

Notice the differences to the output of example "Specifying a bibliography and cite authors"

Read Bibliography in RMD online: https://riptutorial.com/r/topic/7606/bibliography-in-rmd

Chapter 12: boxplot

Syntax

- boxplot(x, ...) # generic function
- boxplot(formula, data = NULL, ..., subset, na.action = NULL) ## S3 method for class 'formula'
- boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE, notch = FALSE, outline = TRUE, names, plot = TRUE, border = par("fg"), col = NULL, log = "", pars = list(boxwex = 0.8, staplewex = 0.5, outwex = 0.5), horizontal = FALSE, add = FALSE, at = NULL) ## Default S3 method

Parameters

Parameters	Details (source R Documentation)
formula	a formula, such as y ~ grp, where y is a numeric vector of data values to be split into groups according to the grouping variable grp (usually a factor).
data	a data.frame (or list) from which the variables in formula should be taken.
subset	an optional vector specifying a subset of observations to be used for plotting.
na.action	a function which indicates what should happen when the data contain NAs. The default is to ignore missing values in either the response or the group.
boxwex	a scale factor to be applied to all boxes. When there are only a few groups, the appearance of the plot can be improved by making the boxes narrower.
plot	if TRUE (the default) then a boxplot is produced. If not, the summaries which the boxplots are based on are returned.
col	if col is non-null it is assumed to contain colors to be used to colour the bodies of the box plots. By default they are in the background colour.

Examples

Create a box-and-whisker plot with boxplot() {graphics}

This example use the default `boxplot()` function and the `iris`data frame.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

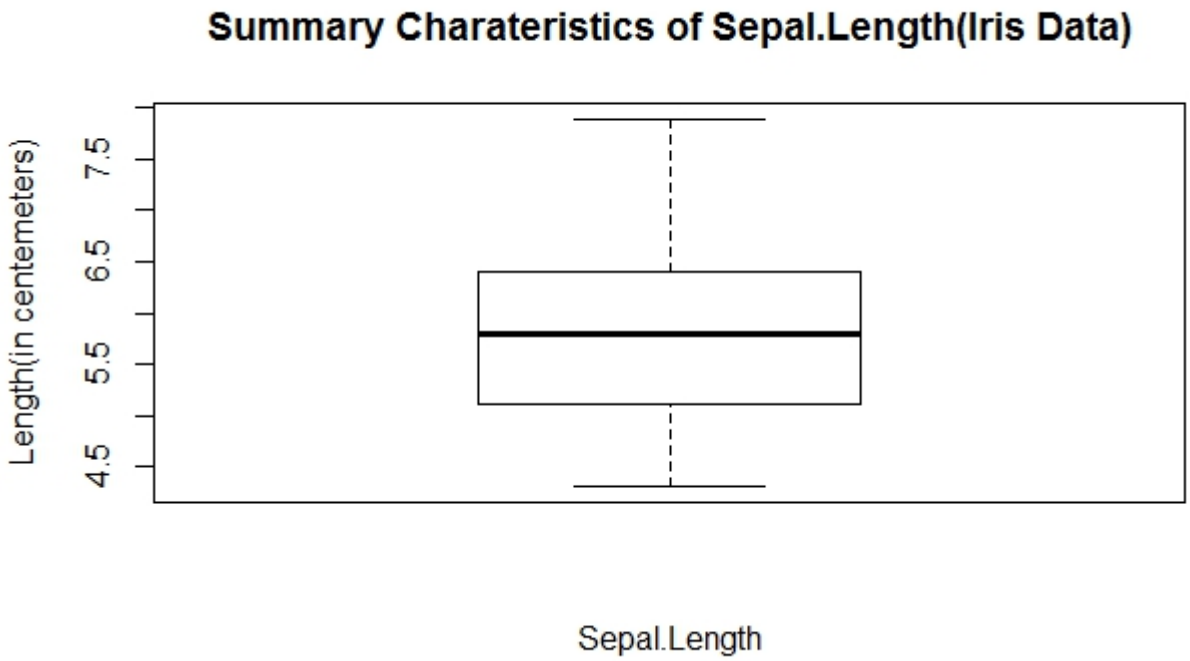

www.dbooks.org

```
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Simple boxplot (Sepal.Length)

Create a box-and-whisker graph of a numerical variable

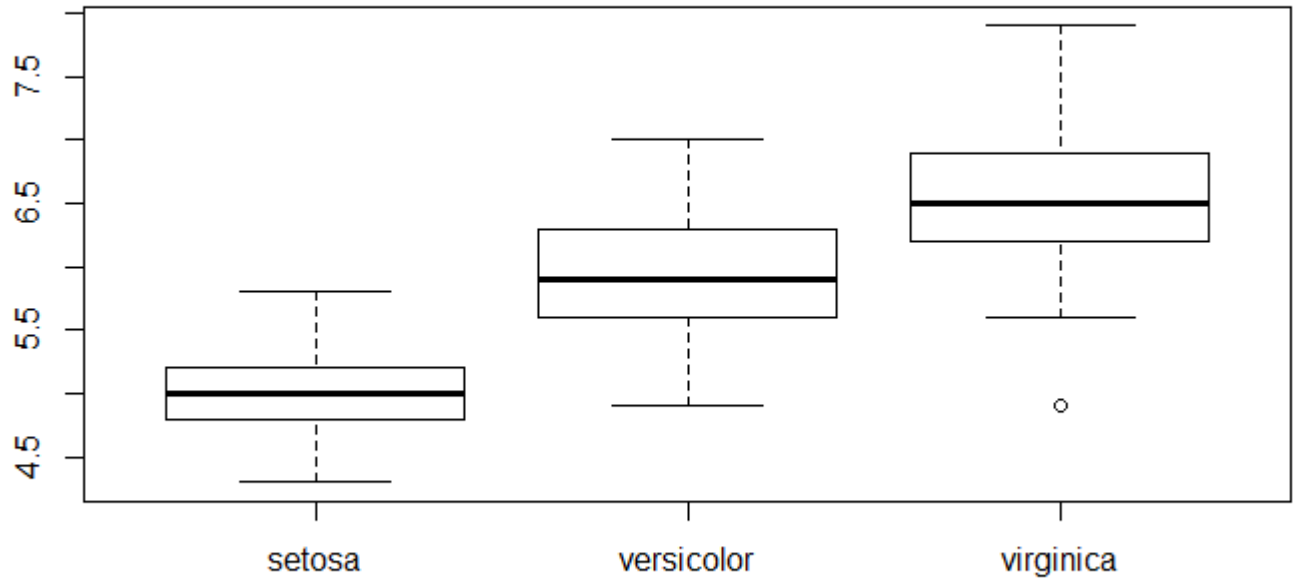
```
boxplot(iris[,1],xlab="Sepal.Length",ylab="Length(in centemeters)",
           main="Summary Charateristics of Sepal.Length(Iris Data)")
```



Boxplot of sepal length grouped by species

Create a boxplot of a numerical variable grouped by a categorical variable

```
boxplot(Sepal.Length~Species,data = iris)
```

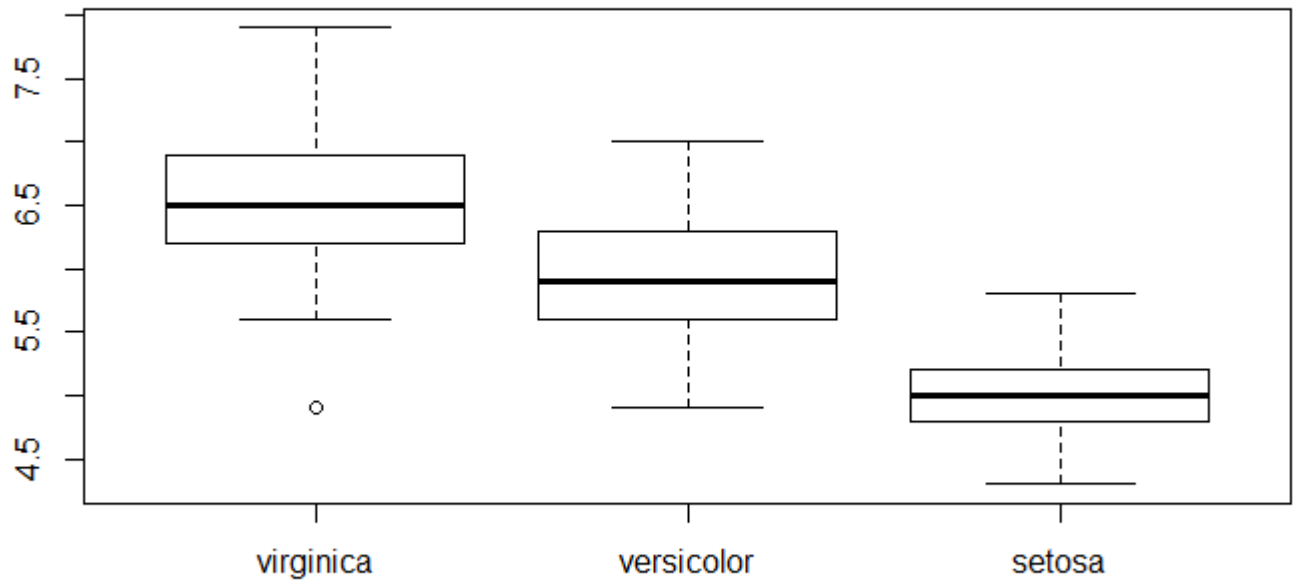


Bring order

To change order of the box in the plot you have to change the order of the categorical variable's levels.
For example if we want to have the order `virginica - versicolor - setosa`

```
newSpeciesOrder <- factor(iris$Species, levels=c("virginica","versicolor","setosa"))
boxplot(Sepal.Length~newSpeciesOrder,data = iris)
```

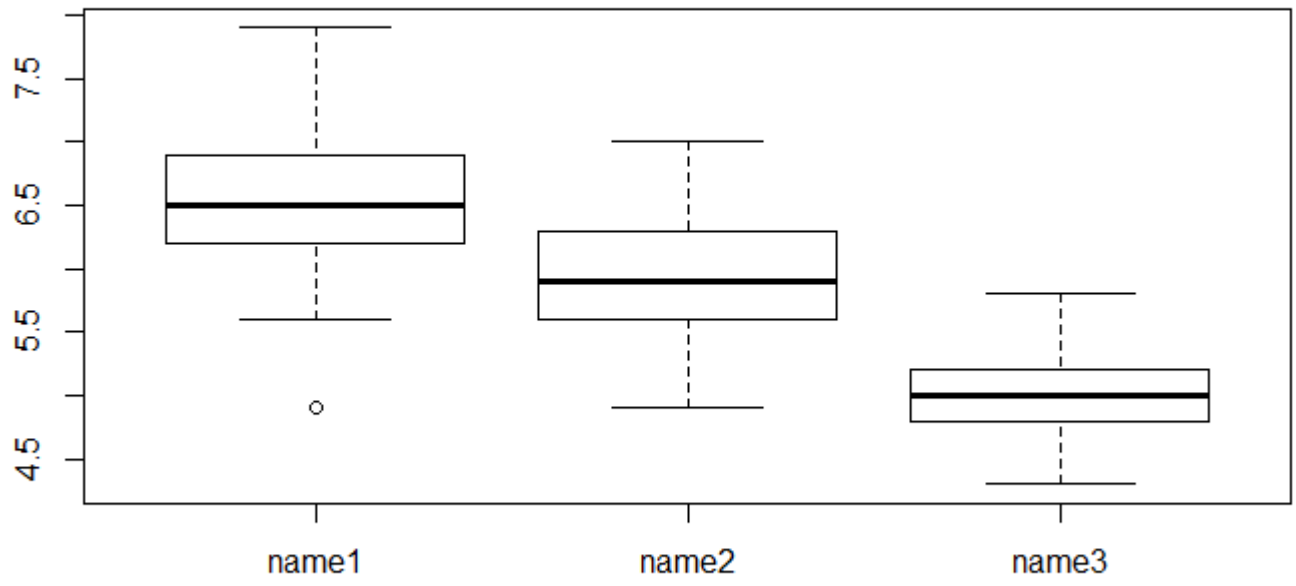

www.dbooks.org



Change groups names

If you want to specifie a better name to your groups you can use the `Names` parameter. It take a vector of the size of the levels of categorical variable

```
boxplot(Sepal.Length~newSpeciesOrder,data = iris,names= c("name1","name2","name3"))
```



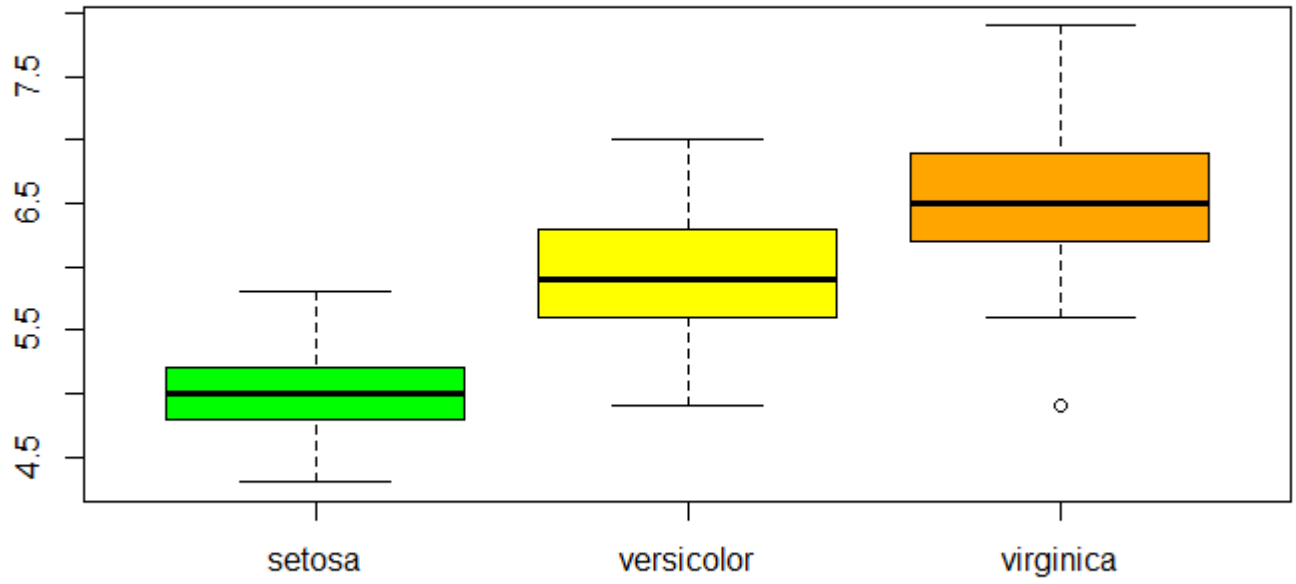
Small improvements

Color

`col` : add a vector of the size of the levels of categorical variable

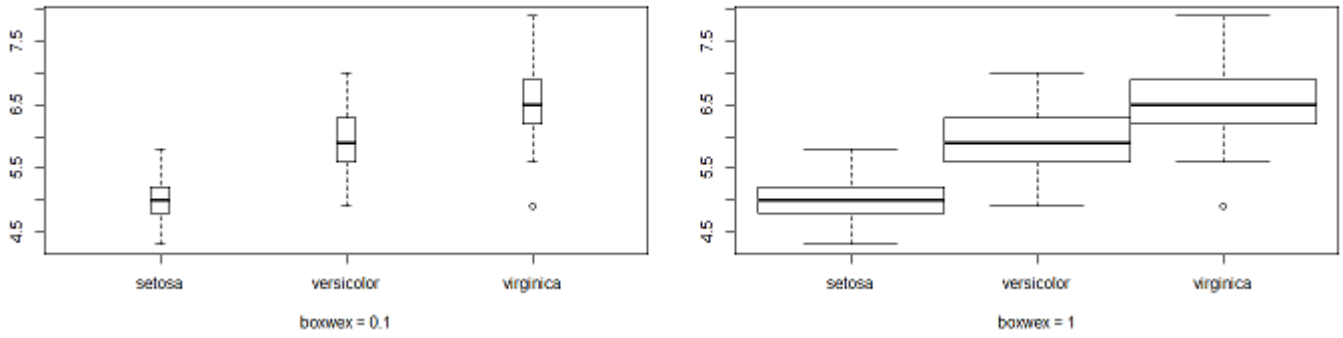
```
boxplot(Sepal.Length~Species,data = iris,col=c("green","yellow","orange"))
```


www.dbooks.org



Proximity of the box

`boxwex`: set the margin between boxes.
Left `boxplot(Sepal.Length~Species,data = iris,boxwex = 0.1)`
Right `boxplot(Sepal.Length~Species,data = iris,boxwex = 1)`



See the summaries which the boxplots are based `plot=FALSE`

To see a summary you have to put the paramater `plot` to `FALSE`.
Various results are given

```
> boxplot(Sepal.Length~newSpeciesOrder,data = iris,plot=FALSE)
$stats #summary of the numerical variable for the 3 groups
     [,1] [,2] [,3]
[1,]  5.6  4.9  4.3 # extreme value
[2,]  6.2  5.6  4.8 # first quartile limit
[3,]  6.5  5.9  5.0 # median limit
[4,]  6.9  6.3  5.2 # third quartile limit
[5,]  7.9  7.0  5.8 # extreme value

$n #number of observations in each groups
[1] 50 50 50

$conf #extreme value of the notchs
         [,1]     [,2]     [,3]
[1,] 6.343588 5.743588 4.910622
[2,] 6.656412 6.056412 5.089378

$out #extreme value
[1] 4.9

$group #group in which are the extreme value
[1] 1

$names #groups names
[1] "virginica"  "versicolor" "setosa"
```

Additional boxplot style parameters.

Box

- boxlty - box line type
- boxlwd - box line width
- boxcol - box line color
- boxfill - box fill colors

Median

- medlty - median line type ("blank" for no line)
- medlwd - median line widht
- medcol - median line color
- medpch - median point (NA for no symbol)
- medcex - median point size
- medbg - median point background color

Whisker

- whisklty - whisker line type
- whisklwd - whisker line width
- whiskcol - whisker line color

www.dbooks.org

Staple

- staplelty - staple line type
- staplelwd - staple line width
- staplecol - staple line color

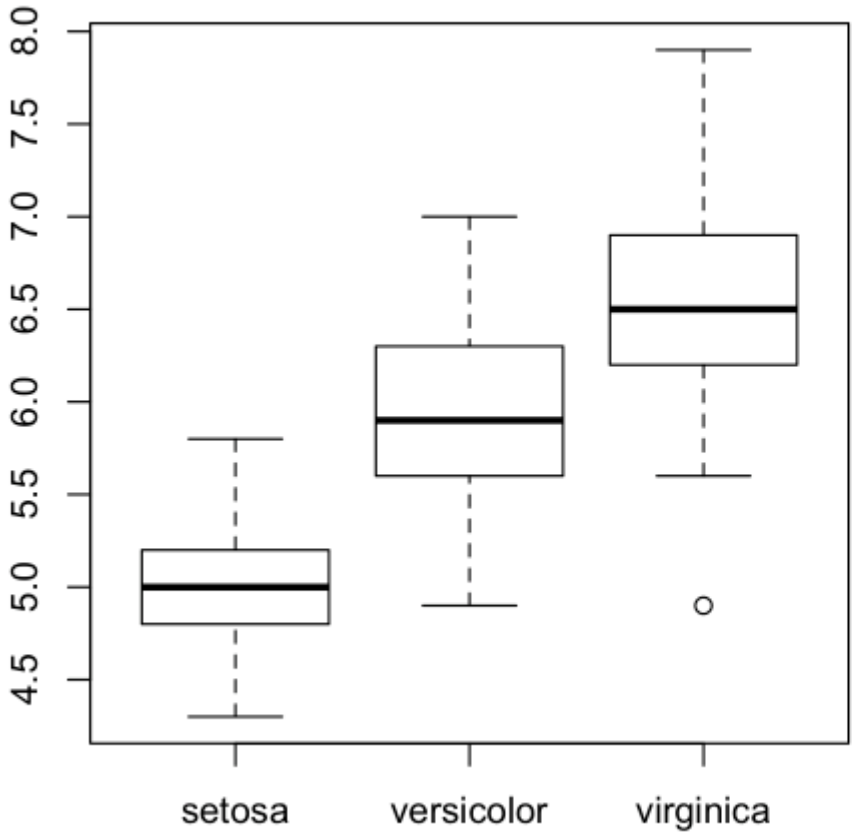
Outliers

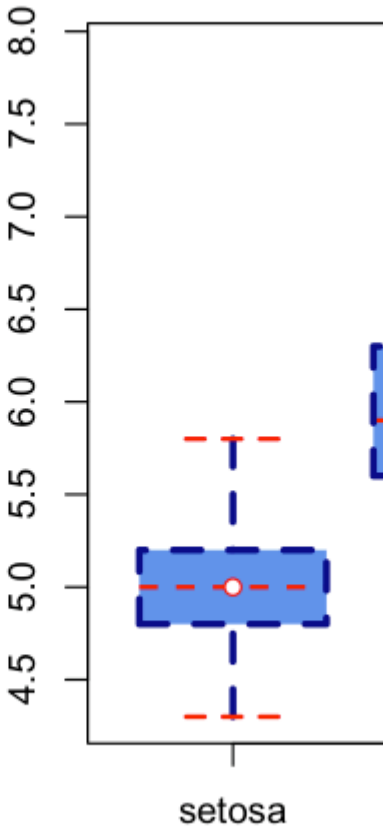
- outlty - outlier line type ("blank" for no line)
- outlwd - outlier line width
- outcol - outlier line color
- outpch - outlier point type (NA for no symbol)
- outcex - outlier point size
- outbg - outlier point background color

Example

Default and heavily modified plots side by side

```
par(mfrow=c(1,2))
# Default
boxplot(Sepal.Length ~ Species, data=iris)
# Modified
boxplot(Sepal.Length ~ Species, data=iris,
        boxlty=2, boxlwd=3, boxfill="cornflowerblue", boxcol="darkblue",
        medlty=2, medlwd=2, medcol="red", medpch=21, medcex=1, medbg="white",
        whisklty=2, whisklwd=3, whiskcol="darkblue",
        staplelty=2, staplelwd=2, staplecol="red",
        outlty=3, outlwd=3, outcol="grey", outpch=NA
        )
```





Read boxplot online: https://riptutorial.com/r/topic/1005/boxplot

www.dbooks.org

Chapter 13: caret

Introduction

`caret` is an R package that aids in data processing needed for machine learning problems. It stands for classification and regression training. When building models for a real dataset, there are some tasks other than the actual learning algorithm that need to be performed, such as cleaning the data, dealing with incomplete observations, validating our model on a test set, and compare different models.

`caret` helps in these scenarios, independent of the actual learning algorithms used.

Examples

Preprocessing

Pre-processing in caret is done through the `preProcess()` function. Given a matrix or data frame type object `x`, `preProcess()` applies transformations on the training data which can then be applied to testing data.

The heart of the `preProcess()` function is the `method` argument. Method operations are applied in this order:

1. Zero-variance filter
2. Near-zero variance filter
3. Box-Cox/Yeo-Johnson/exponential transformation
4. Centering
5. Scaling
6. Range
7. Imputation
8. PCA
9. ICA
10. Spatial Sign

Below, we take the mtcars data set and perform centering, scaling, and a spatial sign transform.

```
auto_index <- createDataPartition(mtcars$mpg, p = .8,
                                  list = FALSE,
                                  times = 1)

mt_train <- mtcars[auto_index,]
mt_test <- mtcars[-auto_index,]

process_mtcars <- preProcess(mt_train, method = c("center","scale","spatialSign"))

mtcars_train_transf <- predict(process_mtcars, mt_train)
mtcars_test_tranf <- predict(process_mtcars,mt_test)
```

Read caret online: https://riptutorial.com/r/topic/4271/caret

www.dbooks.org

Chapter 14: Classes

Introduction

The class of a data-object determines which functions will process its contents. The `class`-attribute is a character vector, and objects can have zero, one or more classes. If there is no class-attribute, there will still be an implicit class determined by an object's `mode`. The class can be inspected with the function `class` and it can be set or modified by the `class<-` function. The S3 class system was established early in S's history. The more complex S4 class system was established later

Remarks

There are several functions for inspecting the "type" of an object. The most useful such function is `class`, although sometimes it is necessary to examine the `mode` of an object. Since we are discussing "types", one might think that `typeof` would be useful, but generally the result from `mode` will be more useful, because objects with no explicit "class"-attribute will have function dispatch determined by the "implicit class" determined by their mode.

Examples

Vectors

The most simple data structure available in R is a vector. You can make vectors of numeric values, logical values, and character strings using the `c()` function. For example:

```
c(1, 2, 3)
## [1] 1 2 3
c(TRUE, TRUE, FALSE)
## [1]  TRUE  TRUE FALSE
c("a", "b", "c")
## [1] "a" "b" "c"
```

You can also join to vectors using the `c()` function.

```
x <- c(1, 2, 5)
y <- c(3, 4, 6)
z <- c(x, y)
z
## [1] 1 2 5 3 4 6
```

A more elaborate treatment of how to create vectors can be found in the *"Creating vectors"* topic

Inspect classes

Every object in R is assigned a class. You can use `class()` to find the object's class and `str()` to

see its structure, including the classes it contains. For example:

```
class(iris)
[1] "data.frame"

str(iris)
'data.frame':    150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 ...

class(iris$Species)
[1] "factor"
```

We see that iris has the class `data.frame` and using `str()` allows us to examine the data inside. The variable Species in the iris data frame is of class factor, in contrast to the other variables which are of class numeric. The `str()` function also provides the length of the variables and shows the first couple of observations, while the `class()` function only provides the object's class.

Vectors and lists

Data in R are stored in vectors. A typical vector is a sequence of values all having the same storage mode (e.g., characters vectors, numeric vectors). See `?atomic` for details on the atomic implicit classes and their corresponding storage modes: `"logical"`, `"integer"`, `"numeric"` (synonym `"double"`), `"complex"`, `"character"` and `"raw"`. Many classes are simply an atomic vector with a `class` attribute on top:

```
x <- 1826
class(x) <- "Date"
x
# [1] "1975-01-01"
 x <- as.Date("1970-01-01")
 class(x)
#[1] "Date"
 is(x,"Date")
#[1] TRUE
 is(x,"integer")
#[1] FALSE
 is(x,"numeric")
#[1] FALSE
  mode(x)
#[1] "numeric"
```

Lists are a special type of vector where each element can be anything, even another list, hence the R term for lists: "recursive vectors":

```
mylist <- list( A = c(5,6,7,8), B = letters[1:10], CC = list( 5, "Z") )
```

Lists have two very important uses:

- Since functions can only return a single value, it is common to return complicated results in a

www.dbooks.org

list:

```
f <- function(x) list(xplus = x + 10, xsq = x^2)

f(7)
# $xplus
# [1] 17
#
# $xsq
# [1] 49
```

- Lists are also the underlying fundamental class for data frames. Under the hood, a data frame is a list of vectors all having the same length:

```
L <- list(x = 1:2, y = c("A","B"))
DF <- data.frame(L)
DF
#   x y
# 1 1 A
# 2 2 B
is.list(DF)
# [1] TRUE
```

The other class of recursive vectors is R expressions, which are "language"- objects

Read Classes online: https://riptutorial.com/r/topic/3563/classes

Chapter 15: Cleaning data

Introduction

Cleaning data in R is paramount to make any analysis. whatever data you have, be it from measurements taken in the field or scraped from the web it is most probable that you will have to reshape it, transform it or filter it to make it suitable for your analysis. In this documentation, we will cover the following topics: - Removing observations with missing data - Factorizing data - Removing incomplete Rows

Examples

Removing missing data from a vector

First lets create a vector called Vector1:

```
set.seed(123)
Vector1 <- rnorm(20)
```

And add missing data to it:

```
set.seed(123)
Vector1[sample(1:length(Vector1), 5)] <- NA
```

Now we can use the is.na function to subset the Vector

```
Vector1 <- Vector1[!is.na(Vector1)]
```

Now the resulting vector will have removed the NAs of the original Vector1

Removing incomplete rows

There might be times where you have a data frame and you want to remove all the rows that might contain an NA value, for that the function *complete.cases* is the best option.

We will use the first 6 rows of the *airquality* dataset to make an example since it already has NAs

```
x <- head(airquality)
```

This has two rows with NAs in the Solar.R column, to remove them we do the following

```
x_no_NA <- x[complete.cases(x),]
```

The resulting dataframe *x_no_NA* will only have complete rows without NAs

www.dbooks.org

Read Cleaning data online: https://riptutorial.com/r/topic/8165/cleaning-data

Chapter 16: Code profiling

Examples

System.time

System time gives you the CPU time required to execute a R expression, for example:

```
system.time(print("hello world"))

# [1] "hello world"
#    user  system elapsed
#       0       0       0
```

You can add larger pieces of code through use of braces:

```
system.time({
    library(numbers)
    Primes(1,10^5)
})
```

Or use it to test functions:

```
fibb <- function (n) {
    if (n < 3) {
        return(c(0,1)[n])
    } else {
        return(fibb(n - 2) + fibb(n -1))
    }
}

system.time(fibb(30))
```

proc.time()

At its simplest, `proc.time()` gives the total elapsed CPU time in seconds for the current process. Executing it in the console gives the following type of output:

```
proc.time()

#       user     system    elapsed
#    284.507    120.397 515029.305
```

This is particularly useful for benchmarking specific lines of code. For example:

```
t1 <- proc.time()
fibb <- function (n) {
    if (n < 3) {
        return(c(0,1)[n])
    } else {
```


www.dbooks.org

```
        return(fibb(n - 2) + fibb(n -1))
    }
}
print("Time one")
print(proc.time() - t1)

t2 <- proc.time()
fibb(30)

print("Time two")
print(proc.time() - t2)
```

This gives the following output:

```
source('~/.active-rstudio-document')

# [1] "Time one"
#    user  system elapsed
#       0       0       0

# [1] "Time two"
#    user  system elapsed
#   1.534   0.012   1.572
```

`system.time()` is a wrapper for `proc.time()` that returns the elapsed time for a particular command/expression.

```
print(t1 <- system.time(replicate(1000,12^2)))
##  user  system elapsed
## 0.000   0.000   0.002
```

Note that the returned object, of class `proc.time`, is slightly more complicated than it appears on the surface:

```
str(t1)
## Class 'proc_time'  Named num [1:5] 0 0 0.002 0 0
##  ..- attr(*, "names")= chr [1:5] "user.self" "sys.self" "elapsed" "user.child" ...
```

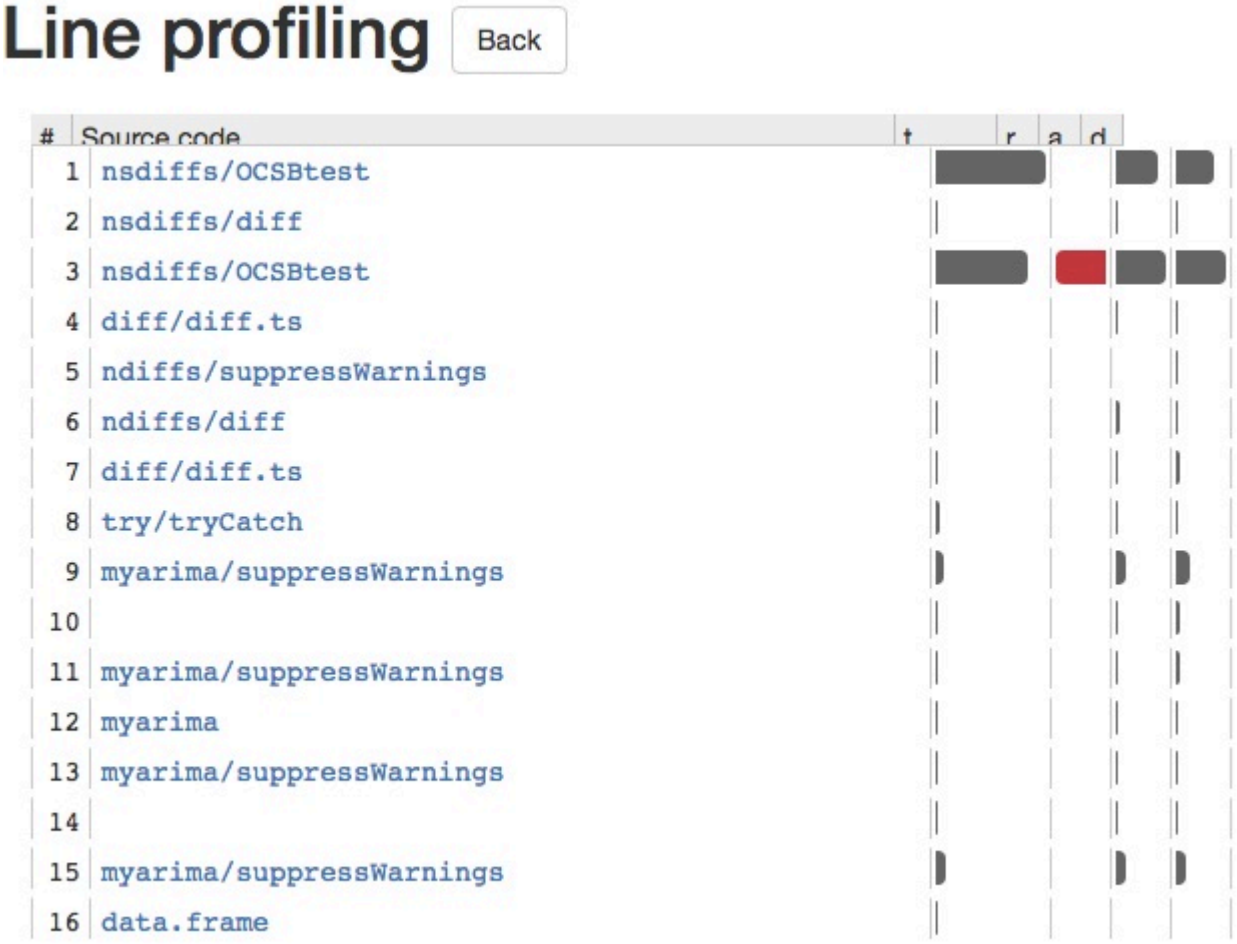
Line Profiling

One package for line profiling is lineprof which is written and maintained by Hadley Wickham. Here is a quick demonstration of how it works with `auto.arima` in the forecast package:

```
library(lineprof)
library(forecast)

l <- lineprof(auto.arima(AirPassengers))
shine(l)
```

This will provide you with a shiny app, which allows you to delve deeper into every function call. This enables you to see with ease what is causing your R code to slow down. There is a screenshot of the shiny app below:



Microbenchmark

Microbenchmark is useful for estimating the time taking for otherwise fast procedures. For example, consider estimating the time taken to print hello world.

```
system.time(print("hello world"))

# [1] "hello world"
#    user  system elapsed
#       0       0       0
```

This is because `system.time` is essentially a wrapper function for `proc.time`, which measures in seconds. As printing "hello world" takes less than a second it appears that the time taken is less than a second, however this is not true. To see this we can use the package microbenchmark:

```
library(microbenchmark)
microbenchmark(print("hello world"))

# Unit: microseconds
#                 expr    min     lq     mean  median     uq     max neval
# print("hello world") 26.336 29.984 44.11637 44.6835 45.415 158.824   100
```

Here we can see after running `print("hello world")` 100 times, the average time taken was in fact

www.dbooks.org

44 microseconds. (Note that running this code will print "hello world" 100 times onto the console.)

We can compare this against an equivalent procedure, `cat("hello world\n")`, to see if it is faster than `print("hello world")`:

```
microbenchmark(cat("hello world\n"))

# Unit: microseconds
#                  expr    min      lq     mean median     uq     max neval
# cat("hello world\\n") 14.093 17.6975 23.73829 19.319 20.996 119.382   100
```

In this case `cat()` is almost twice as fast as `print()`.

Alternatively one can compare two procedures within the same `microbenchmark` call:

```
microbenchmark(print("hello world"), cat("hello world\n"))
# Unit: microseconds
# expr                    min     lq     mean  median     uq     max neval
# print("hello world") 29.122 31.654 39.64255 34.5275 38.852 192.779   100
# cat("hello world\\n")  9.381 12.356 13.83820 12.9930 13.715  52.564   100
```

Benchmarking using microbenchmark

You can use the `microbenchmark` package to conduct "sub-millisecond accurate timing of expression evaluation".

In this example we are comparing the speeds of six equivalent `data.table` expressions for updating elements in a group, based on a certain condition.

More specifically:

> A `data.table` with 3 columns: `id`, `time` and `status`. For each id, I want to find the record with the maximum time - then if for that record if the status is true, I want to set it to false if the time is > 7

```
library(microbenchmark)
library(data.table)

set.seed(20160723)
dt <- data.table(id = c(rep(seq(1:10000), each = 10)),
                 time = c(rep(seq(1:10000), 10)),
                 status = c(sample(c(TRUE, FALSE), 10000*10, replace = TRUE)))
setkey(dt, id, time)  ## create copies of the data so the 'updates-by-reference' don't affect
other expressions
dt1 <- copy(dt)
dt2 <- copy(dt)
dt3 <- copy(dt)
dt4 <- copy(dt)
dt5 <- copy(dt)
dt6 <- copy(dt)

microbenchmark(

  expression_1 = {
```

```
    dt1[ dt1[order(time), .I[.N], by = id]$V1, status := status * time < 7 ]
    },

  expression_2 = {
    dt2[,status := c(.SD[-.N, status], .SD[.N, status * time > 7]), by = id]
    },

  expression_3 = {
    dt3[dt3[,.N, by = id][,cumsum(N)], status := status * time > 7]
    },

  expression_4 = {
    y <- dt4[,.SD[.N],by=id]
    dt4[y, status := status & time > 7]
  },

  expression_5 = {
    y <- dt5[, .SD[.N, .(time, status)], by = id][time > 7 & status]
    dt5[y, status := FALSE]
  },

  expression_6 = {
    dt6[ dt6[, .I == .I[which.max(time)], by = id]$V1 & time > 7, status := FALSE]
    },

  times = 10L ## specify the number of times each expression is evaluated
)

# Unit: milliseconds
#          expr          min           lq         mean       median          uq           max neval
# expression_1    11.646149    13.201670    16.808399    15.643384    18.78640     26.321346    10
# expression_2 8051.898126 8777.016935 9238.323459 8979.553856 9281.93377 12610.869058    10
# expression_3     3.208773     3.385841     4.207903     4.089515     4.70146      5.654702    10
# expression_4    15.758441    16.247833    20.677038    19.028982    21.04170     36.373153    10
# expression_5 7552.970295 8051.080753 8702.064620 8861.608629 9308.62842  9722.234921    10
# expression_6    18.403105    18.812785    22.427984    21.966764    24.66930     28.607064    10
```

The output shows that in this test `expression_3` is the fastest.

References

data.table - Adding and modifying columns

data.table - special grouping symbols in data.table

Read Code profiling online: https://riptutorial.com/r/topic/2149/code-profiling

www.dbooks.org

Chapter 17: Coercion

Introduction

Coercion happens in R when the type of objects are changed during computation either implicitly or by using functions for explicit coercion (such as as.numeric, as.data.frame, etc.).

Examples

Implicit Coercion

Coercion happens with data types in R, often implicitly, so that the data can accommodate all the values. For example,

```
x = 1:3
x
[1] 1 2 3
typeof(x)
#[1] "integer"

x[2] = "hi"
x
#[1] "1"  "hi" "3"
typeof(x)
#[1] "character"
```

Notice that at first, `x` is of type `integer`. But when we assigned `x[2] = "hi"`, all the elements of `x` were coerced into `character` as vectors in R can only hold data of single type.

Read Coercion online: https://riptutorial.com/r/topic/9793/coercion

Chapter 18: Color schemes for graphics

Examples

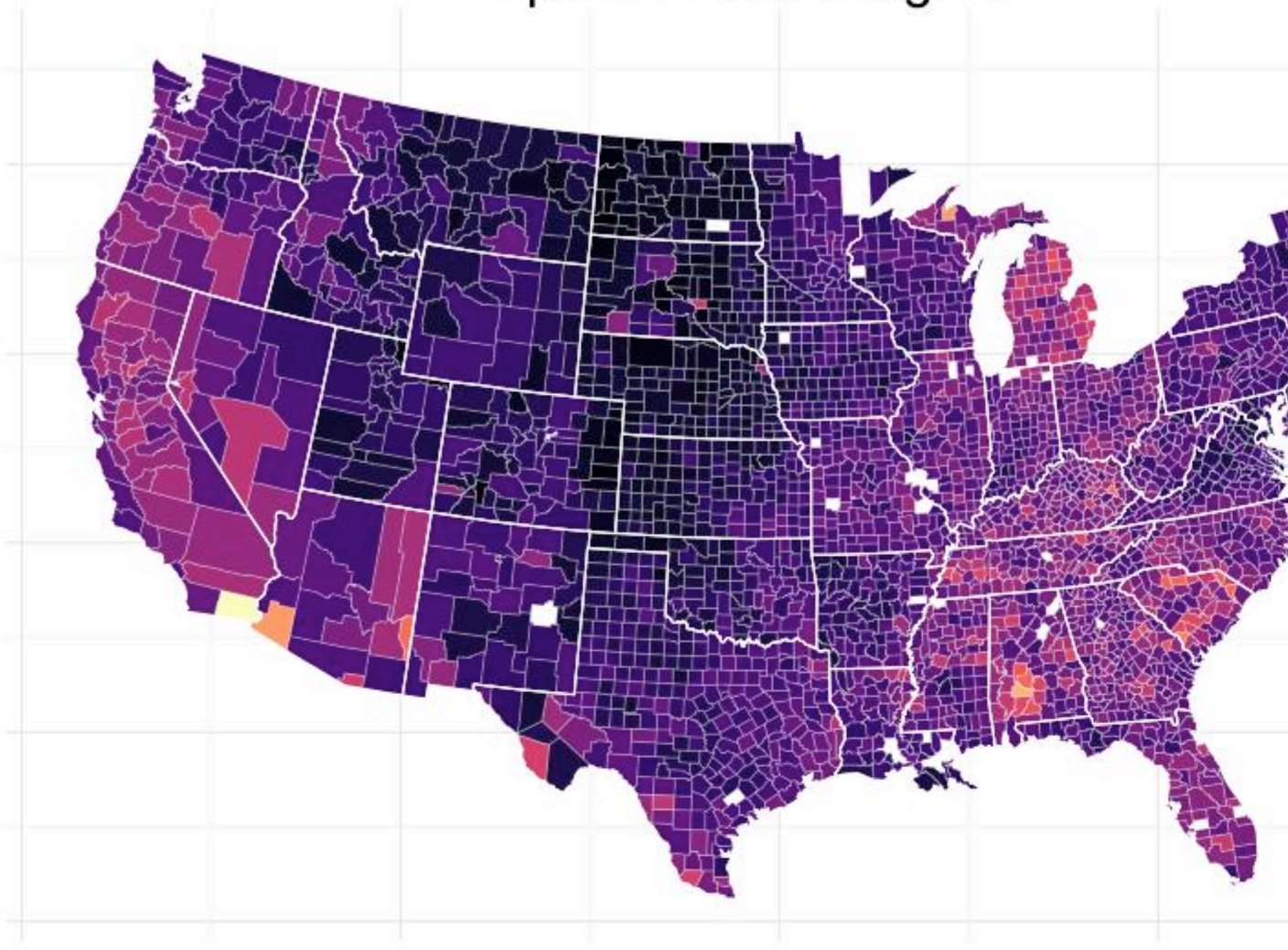
viridis - print and colorblind friendly palettes

Viridis (named after the chromis viridis fish) is a recently developed color scheme for the Python library `matplotlib` (the video presentation by the link explains how the color scheme was developed and what are its main advantages). It is seamlessly ported to `R`.

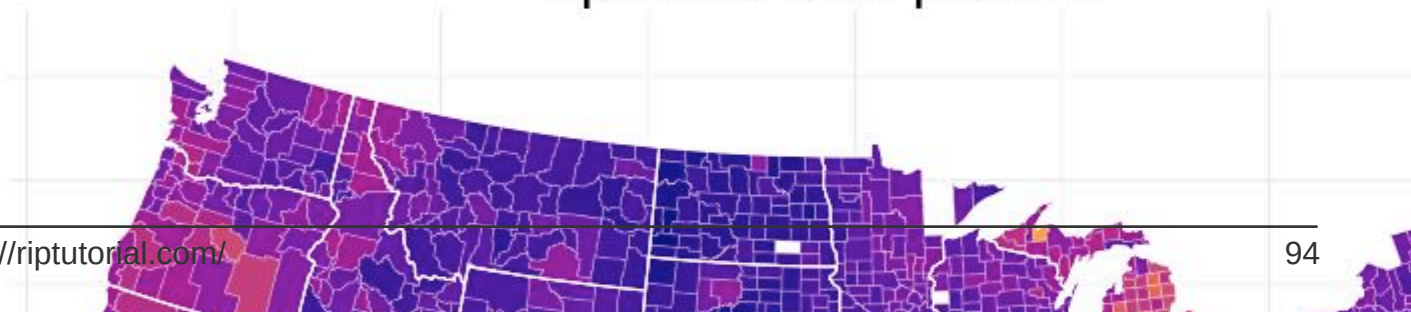
There are 4 variants of color schemes: `magma`, `plasma`, `inferno`, and `viridis` (default). They are chosen with the `option` parameter and are coded as `A`, `B`, `C`, and `D`, correspondingly. To have an impression of the 4 color schemes, look at the maps:

U

option A aka 'magma'



option C aka 'plasma'



(image souce)

The package can be installed from CRAN or github.

The vignette for `viridis` package is just brilliant.

Nice feature of the `viridis` color scheme is integration with `ggplot2`. Within the package two `ggplot2` -specific functions are defined: `scale_color_viridis()` and `scale_fill_viridis()`. See the example below:

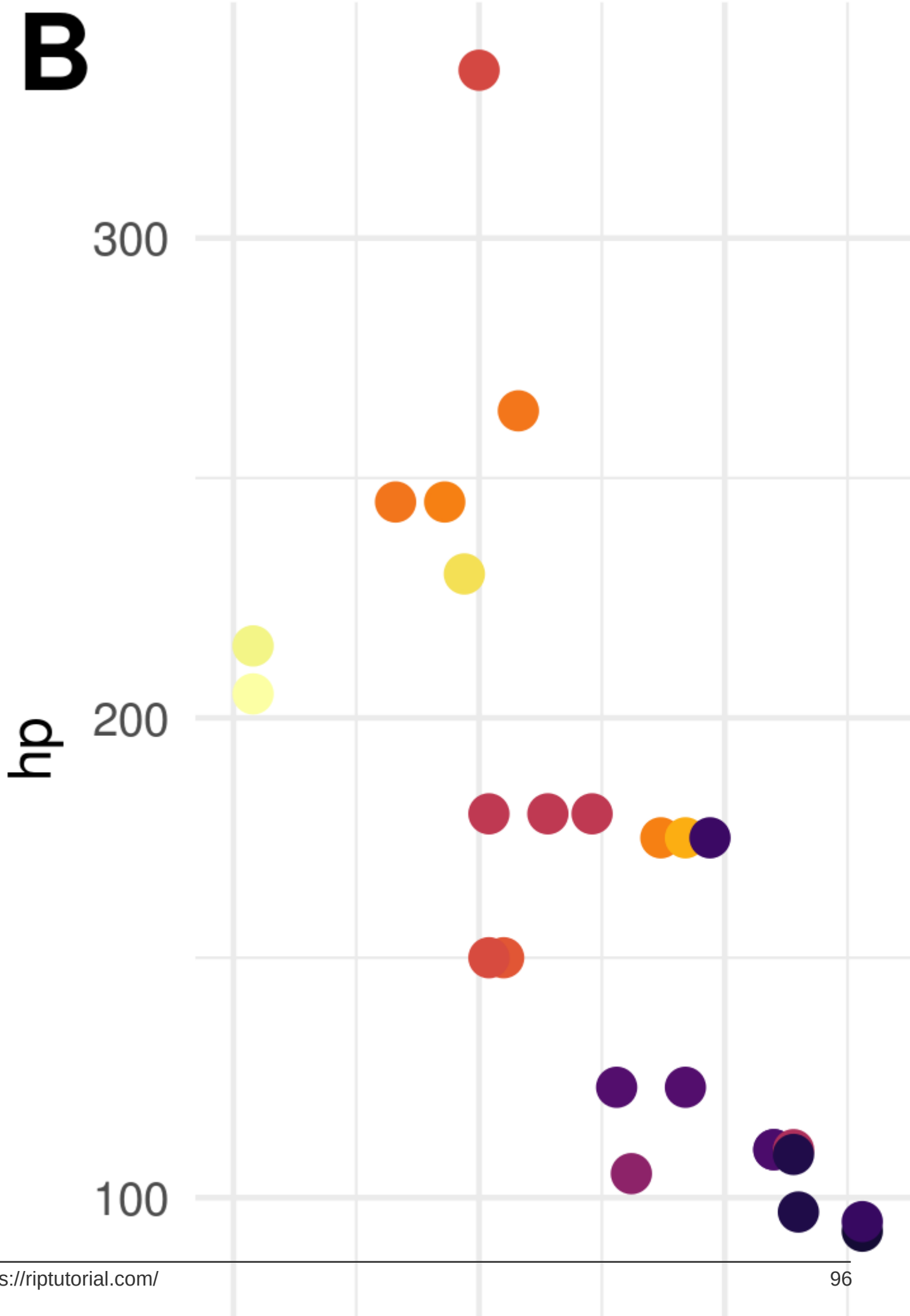
```
library(viridis)
library(ggplot2)

gg1 <- ggplot(mtcars)+
    geom_point(aes(x = mpg, y = hp, color = disp), size = 3)+
    scale_color_viridis(option = "B")+
    theme_minimal()+
    theme(legend.position = c(.8,.8))

gg2 <- ggplot(mtcars)+
        geom_violin(aes(x = factor(cyl), y = hp, fill = factor(cyl)))+
        scale_fill_viridis(discrete = T)+
        theme_minimal()+
        theme(legend.position = 'none')

library(cowplot)
output <- plot_grid(gg1,gg2, labels = c('B','D'),label_size = 20)
print(output)
```


www.dbooks.org


B
300
200
100
hp

`RColorBrewer` is a port of the project for `R` and provides also colorblind-friendly palettes.

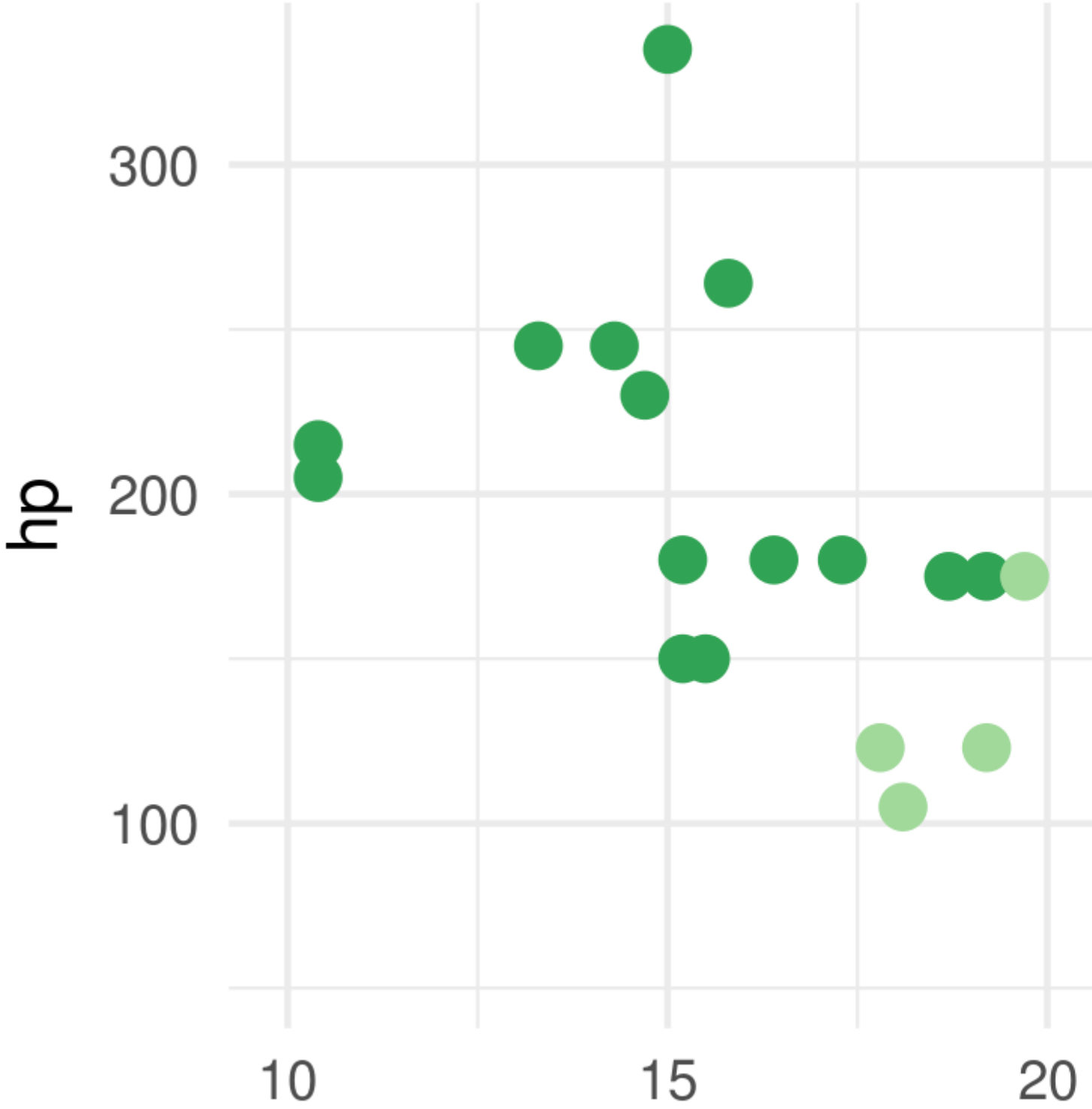
An example of use

```
colors_vec <- brewer.pal(5, name = 'BrBG')
print(colors_vec)
[1] "#A6611A" "#DFC27D" "#F5F5F5" "#80CDC1" "#018571"
```

`RColorBrewer` creates coloring options for `ggplot2`: `scale_color_brewer` and `scale_fill_brewer`.

```
library(ggplot2)
ggplot(mtcars)+
        geom_point(aes(x = mpg, y = hp, color = factor(cyl)), size = 3)+
        scale_color_brewer(palette = 'Greens')+
        theme_minimal()+
        theme(legend.position = c(.8,.8))
```


www.dbooks.org



A handy function to glimse a vector of colors

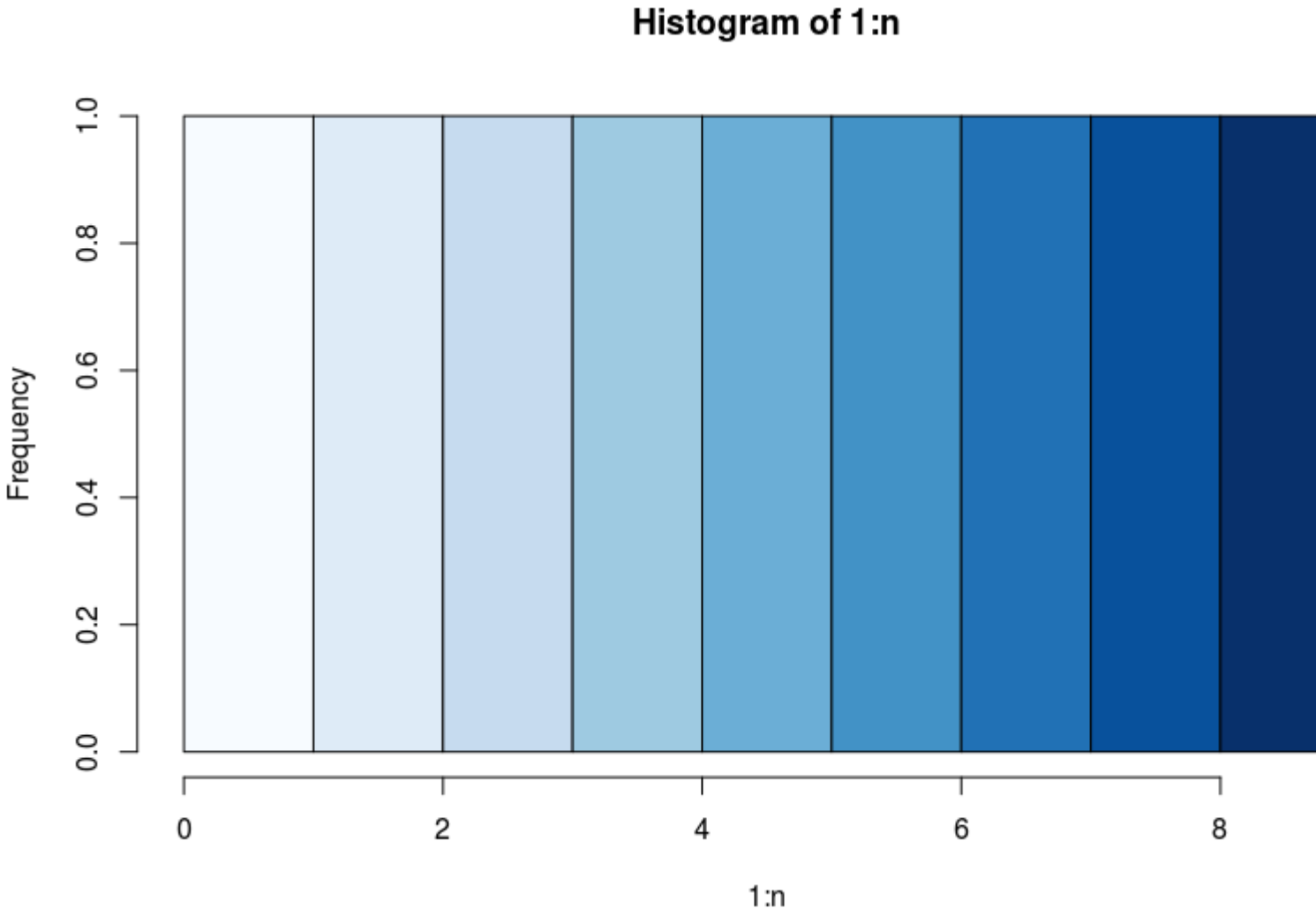
Quite often there is a need to glimpse the chosen color palette. One elegant solution is the following self defined function:

```
color_glimpse <- function(colors_string){
```

```
        n <- length(colors_string)
        hist(1:n,breaks=0:n,col=colors_string)
}
```

An example of use

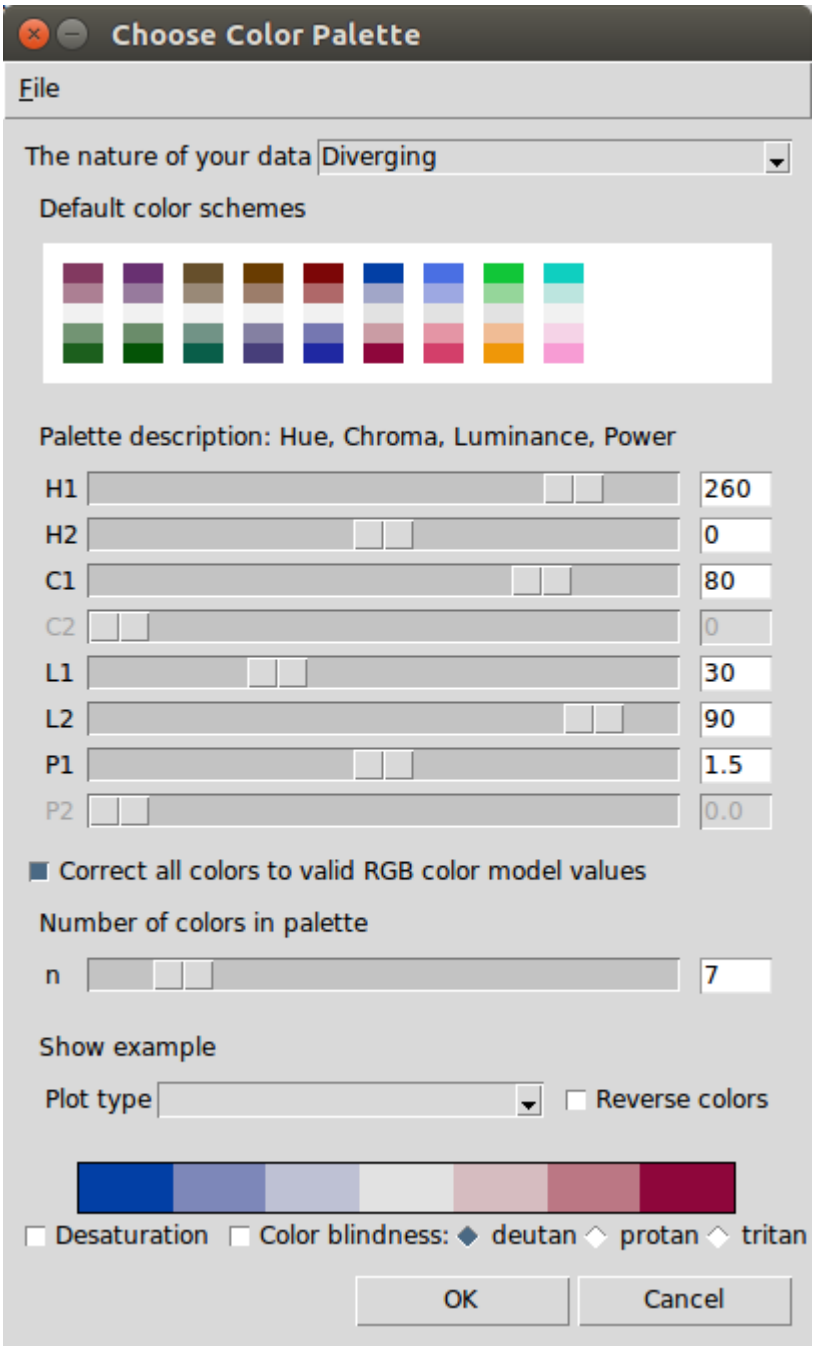
```
color_glimpse(blues9)
```



colorspace - click&drag interface for colors

The package `colorspace` provides GUI for selecting a palette. On the call of `choose_palette()` function the following window pops-up:

www.dbooks.org



When the palette is chosen, just hit `OK` and do not forget to store the output in a variable, e.g. `pal`.

```
pal <- choose_palette()
```

The output is a function that takes `n` (number) as input and produces a color vector of length `n` according to the selected palette.

```
pal(10)
[1] "#023FA5" "#6371AF" "#959CC3" "#BEC1D4" "#DBDCE0" "#E0DBDC" "#D6BCC0" "#C6909A" "#AE5A6D"
"#8E063B"
```

basic R color functions

Function `colors()` lists all the color names that are recognized by R. There is a nice PDF where

one can actually see those colors.

`colorRampPalette` creates a function that interpolate a set of given colors to create new color palettes. This output function takes `n` (number) as input and produces a color vector of length `n` interpolating the initial colors.

```
pal <- colorRampPalette(c('white','red'))
pal(5)
[1] "#FFFFFF" "#FFBFBF" "#FF7F7F" "#FF3F3F" "#FF0000"
```

Any specific color may be produced with an `rgb()` function:

```
rgb(0,1,0)
```

produces `green` color.

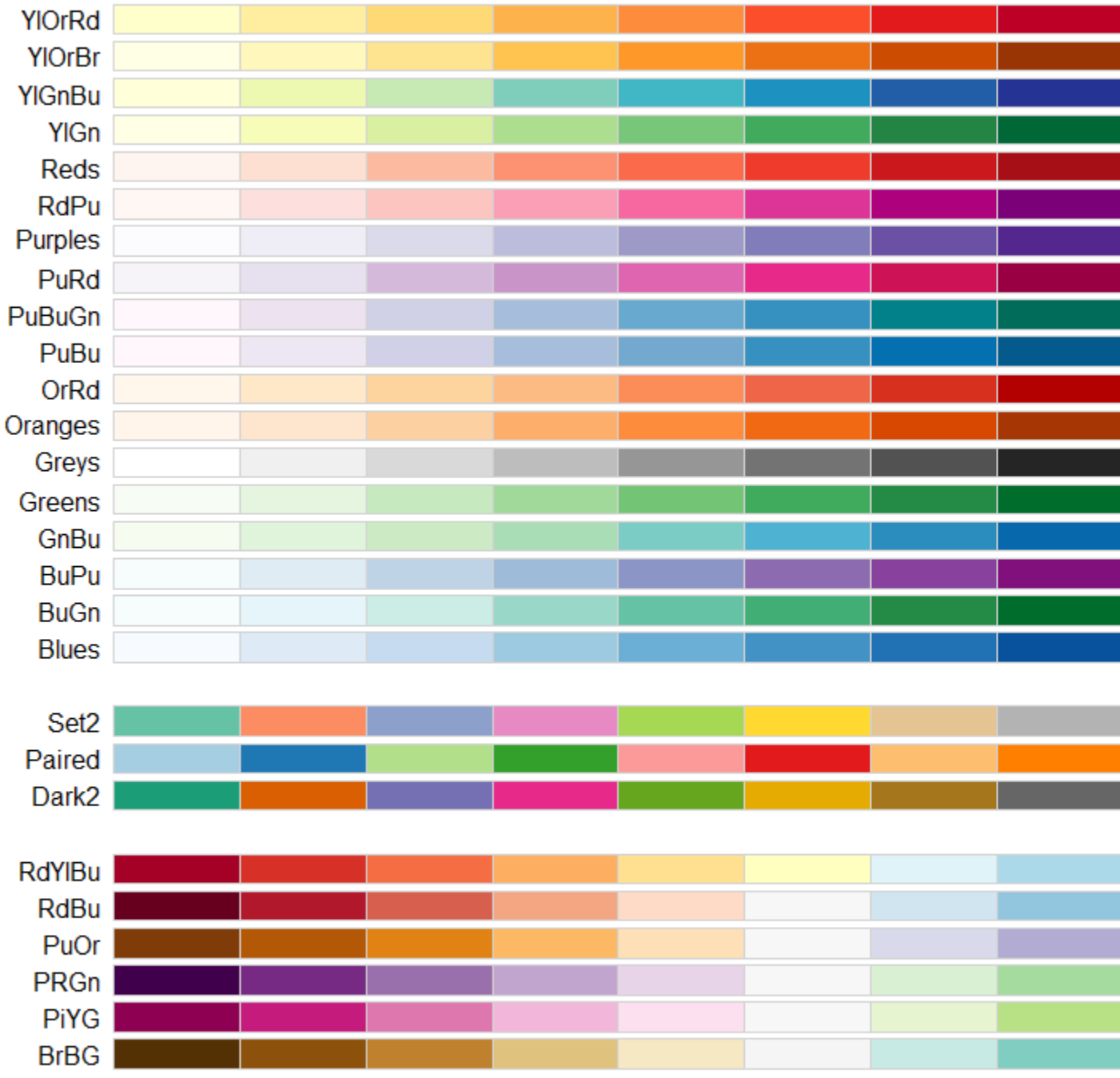
Colorblind-friendly palettes

Even though colorblind people can recognize a wide range of colors, it might be hard to differentiate between certain colors.

`RColorBrewer` provides colorblind-friendly palettes:

```
library(RColorBrewer)
display.brewer.all(colorblindFriendly = T)
```


www.dbooks.org



The Color Universal Design from the University of Tokyo proposes the following palettes:

```
#palette using grey
```

```
cbPalette <- c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00",
"#CC79A7")

#palette using black
cbbPalette <- c("#000000", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00",
"#CC79A7")
```

Read Color schemes for graphics online: https://riptutorial.com/r/topic/8005/color-schemes-for-graphics

www.dbooks.org

Chapter 19: Column wise operation

Examples

sum of each column

Suppose we need to do the `sum` of each column in a dataset

```
set.seed(20)
df1 <- data.frame(ID = rep(c("A", "B", "C"), each = 3), V1 = rnorm(9), V2 = rnorm(9))
m1 <- as.matrix(df1[-1])
```

There are many ways to do this. Using `base R`, the best option would be `colSums`

```
colSums(df1[-1], na.rm = TRUE)
```

Here, we removed the first column as it is non-numeric and did the `sum` of each column, specifying the `na.rm = TRUE` (in case there are any NAs in the dataset)

This also works with `matrix`

```
colSums(m1, na.rm = TRUE)
```

This can be done in a loop with `lapply/sapply/vapply`

```
lapply(df1[-1], sum, na.rm = TRUE)
```

It should be noted that the output is a `list`. If we need a `vector` output

```
sapply(df1[-1], sum, na.rm = TRUE)
```

Or

```
vapply(df1[-1], sum, na.rm = TRUE, numeric(1))
```

For matrices, if we want to loop through columns, then use `apply` with `MARGIN = 1`

```
apply(m1, 2, FUN = sum, na.rm = TRUE)
```

There are ways to do this with packages like `dplyr` or `data.table`

```
library(dplyr)
df1 %>%
    summarise_at(vars(matches("^V\\d+")), sum, na.rm = TRUE)
```

Here, we are passing a regular expression to match the column names that we need to get the `sum` in `summarise_at`. The regex will match all columns that start with `v` followed by one or more numbers (`\\d+`).

A `data.table` option is

```
library(data.table)
setDT(df1)[, lapply(.SD, sum, na.rm = TRUE), .SDcols = 2:ncol(df1)]
```

We convert the 'data.frame' to 'data.table' (`setDT(df1)`), specified the columns to be applied the function in `.SDcols` and loop through the Subset of Data.table (`.SD`) and get the `sum`.

If we need to use a group by operation, we can do this easily by specifying the group by column/columns

```
 df1 %>%
   group_by(ID) %>%
   summarise_at(vars(matches("^V\\d+")), sum, na.rm = TRUE)
```

In cases where we need the `sum` of all the columns, `summarise_each` can be used instead of `summarise_at`

```
df1 %>%
    group_by(ID) %>%
    summarise_each(funs(sum(., na.rm = TRUE)))
```

The `data.table` option is

```
setDT(df1)[, lapply(.SD, sum, na.rm = TRUE), by = ID]
```

Read Column wise operation online: https://riptutorial.com/r/topic/2212/column-wise-operation

www.dbooks.org

Chapter 20: Combinatorics

Examples

Enumerating combinations of a specified length

Without replacement

With `combn`, each vector appears in a column:

```
combn(LETTERS, 3)

# Showing only first 10.
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"
[2,] "B"  "B"  "B"  "B"  "B"  "B"  "B"  "B"  "B"  "B"
[3,] "C"  "D"  "E"  "F"  "G"  "H"  "I"  "J"  "K"  "L"
```

With replacement

With `expand.grid`, each vector appears in a row:

```
expand.grid(LETTERS, LETTERS, LETTERS)
# or
do.call(expand.grid, rep(list(LETTERS), 3))

# Showing only first 10.
   Var1 Var2 Var3
1     A    A    A
2     B    A    A
3     C    A    A
4     D    A    A
5     E    A    A
6     F    A    A
7     G    A    A
8     H    A    A
9     I    A    A
10    J    A    A
```

For the special case of pairs, `outer` can be used, putting each vector into a cell:

```
# FUN here is used as a function executed on each resulting pair.
# in this case it's string concatenation.
outer(LETTERS, LETTERS, FUN=paste0)

# Showing only first 10 rows and columns
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,] "AA" "AB" "AC" "AD" "AE" "AF" "AG" "AH" "AI" "AJ"
 [2,] "BA" "BB" "BC" "BD" "BE" "BF" "BG" "BH" "BI" "BJ"
```

```
 [3,] "CA" "CB" "CC" "CD" "CE" "CF" "CG" "CH" "CI" "CJ"
 [4,] "DA" "DB" "DC" "DD" "DE" "DF" "DG" "DH" "DI" "DJ"
 [5,] "EA" "EB" "EC" "ED" "EE" "EF" "EG" "EH" "EI" "EJ"
 [6,] "FA" "FB" "FC" "FD" "FE" "FF" "FG" "FH" "FI" "FJ"
 [7,] "GA" "GB" "GC" "GD" "GE" "GF" "GG" "GH" "GI" "GJ"
 [8,] "HA" "HB" "HC" "HD" "HE" "HF" "HG" "HH" "HI" "HJ"
 [9,] "IA" "IB" "IC" "ID" "IE" "IF" "IG" "IH" "II" "IJ"
[10,] "JA" "JB" "JC" "JD" "JE" "JF" "JG" "JH" "JI" "JJ"
```

Counting combinations of a specified length

Without replacement

```
choose(length(LETTERS), 5)
[1] 65780
```

With replacement

```
length(letters)^5
[1] 11881376
```

Read Combinatorics online: https://riptutorial.com/r/topic/5836/combinatorics

www.dbooks.org

Chapter 21: Control flow structures

Remarks

For loops are a flow control method for repeating a task or set of tasks over a domain. The core structure of a for loop is

```
for ( [index] in [domain]){
  [body]
}
```

Where

1. `[index]` is a name takes exactly one value of `[domain]` over each iteration of the loop.
2. `[domain]` is a vector of values over which to iterate.
3. `[body]` is the set of instructions to apply on each iteration.

As a trivial example, consider the use of a for loop to obtain the cumulative sum of a vector of values.

```
x <- 1:4
cumulative_sum <- 0
for (i in x){
  cumulative_sum <- cumulative_sum + x[i]
}
cumulative_sum
```

Optimizing Structure of For Loops

For loops can be useful for conceptualizing and executing tasks to repeat. If not constructed carefully, however, they can be very slow to execute compared to the preferred used of the `apply` family of functions. Nonetheless, there are a handful of elements you can include in your for loop construction to optimize the loop. In many cases, good construction of the for loop will yield computational efficiency very close to that of an apply function.

A 'properly constructed' for loop builds on the core structure and includes a statement declaring the object that will capture each iteration of the loop. This object should have both a class and a length declared.

```
[output] <- [vector_of_length]
for ([index] in [length_safe_domain]){
  [output][index] <- [body]
}
```

To illustrate, let us write a loop to square each value in a numeric vector (this is a trivial example for illustration only. The 'correct' way of completing this task would be `x_squared <- x^2`).

```
x <- 1:100
x_squared <- vector("numeric", length = length(x))
for (i in seq_along(x)){
  x_squared[i] <- x[i]^2
}
```

Again, notice that we first declared a receptacle for the output `x_squared`, and gave it the class "numeric" with the same length as `x`. Additionally, we declared a "length safe domain" using the `seq_along` function. `seq_along` generates a vector of indices for an object that is suited for use in for loops. While it seems intuitive to use `for (i in 1:length(x))`, if `x` has 0 length, the loop will attempt to iterate over the domain of `1:0`, resulting in an error (the 0th index is undefined in R).

Receptacle objects and length safe domains are handled internally by the `apply` family of functions and users are encouraged to adopt the `apply` approach in place of for loops as much as possible. However, if properly constructed, a for loop may occasionally provide greater code clarity with minimal loss of efficiency.

Vectorizing For Loops

For loops can often be a useful tool in conceptualizing the tasks that need to be completed within each iteration. When the loop is completely developed and conceptualized, there may be advantages to turning the loop into a function.

In this example, we will develop a for loop to calculate the mean of each column in the `mtcars` dataset (again, a trivial example as it could be accomplished via the `colMeans` function).

```
column_mean_loop <- vector("numeric", length(mtcars))
for (k in seq_along(mtcars)){
  column_mean_loop[k] <- mean(mtcars[[k]])
}
```

The for loop can be converted to an apply function by rewriting the body of the loop as a function.

```
col_mean_fn <- function(x) mean(x)
column_mean_apply <- vapply(mtcars, col_mean_fn, numeric(1))
```

And to compare the results:

```
identical(column_mean_loop,
          unname(column_mean_apply)) #* vapply added names to the elements
                                     #* remove them for comparison
```

The advantages of the vectorized form is that we were able to eliminate a few lines of code. The mechanics of determining the length and type of the output object and iterating over a length safe domain are handled for us by the apply function. Additionally, the apply function is a little bit faster than the loop. The difference of speed is often negligible in human terms depending on the number of iterations and the complexity of the body.

www.dbooks.org

Examples

Basic For Loop Construction

In this example we will calculate the squared deviance for each column in a data frame, in this case the `mtcars`.

Option A: integer index

```
squared_deviance <- vector("list", length(mtcars))
for (i in seq_along(mtcars)){
  squared_deviance[[i]] <- (mtcars[[i]] - mean(mtcars[[i]]))^2
}
```

`squared_deviance` is an 11 elements list, as expected.

```
class(squared_deviance)
length(squared_deviance)
```

Option B: character index

```
squared_deviance <- vector("list", length(mtcars))
Squared_deviance <- setNames(squared_deviance, names(mtcars))
for (k in names(mtcars)){
  squared_deviance[[k]] <- (mtcars[[k]] - mean(mtcars[[k]]))^2
}
```

What if we want a `data.frame` as a result? Well, there are many options for transforming a list into other objects. However, and maybe the simplest in this case, will be to store the `for` results in a `data.frame`.

```
squared_deviance <- mtcars #copy the original
squared_deviance[TRUE]<-NA  #replace with NA or do squared_deviance[,]<-NA
for (i in seq_along(mtcars)){
  squared_deviance[[i]] <- (mtcars[[i]] - mean(mtcars[[i]]))^2
}
dim(squared_deviance)
[1] 32 11
```

The result will be the same event though we use the character option (B).

Optimal Construction of a For Loop

To illustrate the effect of good for loop construction, we will calculate the mean of each column in four different ways:

1. Using a poorly optimized for loop
2. Using a well optimized for for loop
3. Using an `*apply` family of functions
4. Using the `colMeans` function

Each of these options will be shown in code; a comparison of the computational time to execute each option will be shown; and lastly a discussion of the differences will be given.

Poorly optimized for loop

```
column_mean_poor <- NULL
for (i in 1:length(mtcars)){
  column_mean_poor[i] <- mean(mtcars[[i]])
}
```

Well optimized for loop

```
column_mean_optimal <- vector("numeric", length(mtcars))
for (i in seq_along(mtcars)){
  column_mean_optimal <- mean(mtcars[[i]])
}
```

`vapply` Function

```
column_mean_vapply <- vapply(mtcars, mean, numeric(1))
```

`colMeans` Function

```
column_mean_colMeans <- colMeans(mtcars)
```

Efficiency comparison

The results of benchmarking these four approaches is shown below (code not displayed)

```
Unit: microseconds
     expr     min       lq     mean   median       uq     max neval  cld
     poor 240.986 262.0820 287.1125 275.8160 307.2485 442.609   100    d
  optimal 220.313 237.4455 258.8426 247.0735 280.9130 362.469   100   c
   vapply 107.042 109.7320 124.4715 113.4130 132.6695 202.473   100 a
 colMeans 155.183 161.6955 180.2067 175.0045 194.2605 259.958   100  b
```

Notice that the optimized `for` loop edged out the poorly constructed for loop. The poorly constructed for loop is constantly increasing the length of the output object, and at each change of the length, R is reevaluating the class of the object.

Some of this overhead burden is removed by the optimized for loop by declaring the type of output object and its length before starting the loop.

In this example, however, the use of an `vapply` function doubles the computational efficiency, largely because we told R that the result had to be numeric (if any one result were not numeric, an error would be returned).

www.dbooks.org

Use of the `colMeans` function is a touch slower than the `vapply` function. This difference is attributable to some error checks performed in `colMeans` and mainly to the `as.matrix` conversion (because `mtcars` is a `data.frame`) that weren't performed in the `vapply` function.

The Other Looping Constructs: while and repeat

R provides two additional looping constructs, `while` and `repeat`, which are typically used in situations where the number of iterations required is indeterminate.

The `while` loop

The general form of a `while` loop is as follows,

```
while (condition) {
    ## do something
    ## in loop body
}
```

where `condition` is evaluated prior to entering the loop body. If `condition` evaluates to `TRUE`, the code inside of the loop body is executed, and this process repeats until `condition` evaluates to `FALSE` (or a `break` statement is reached; see below). Unlike the `for` loop, if a `while` loop uses a variable to perform incremental iterations, the variable must be declared and initialized ahead of time, and must be updated within the loop body. For example, the following loops accomplish the same task:

```
for (i in 0:4) {
    cat(i, "\n")
}
# 0
# 1
# 2
# 3
# 4

i <- 0
while (i < 5) {
    cat(i, "\n")
    i <- i + 1
}
# 0
# 1
# 2
# 3
# 4
```

In the `while` loop above, the line `i <- i + 1` is necessary to prevent an infinite loop.

Additionally, it is possible to terminate a `while` loop with a call to `break` from inside the loop body:

```
iter <- 0
```

```
while (TRUE) {
    if (runif(1) < 0.25) {
        break
    } else {
        iter <- iter + 1
    }
}
iter
#[1] 4
```

In this example, `condition` is always `TRUE`, so the only way to terminate the loop is with a call to `break` inside the body. Note that the final value of `iter` will depend on the state of your PRNG when this example is run, and should produce different results (essentially) each time the code is executed.

The `repeat` loop

The `repeat` construct is essentially the same as `while (TRUE) { ## something }`, and has the following form:

```
repeat ({
    ## do something
    ## in loop body
})
```

The extra `{}` are not required, but the `()` are. Rewriting the previous example using `repeat`,

```
iter <- 0
repeat ({
    if (runif(1) < 0.25) {
        break
    } else {
        iter <- iter + 1
    }
})
iter
#[1] 2
```

More on `break`

It's important to note that `break` will *only terminate the immediately enclosing loop*. That is, the following is an infinite loop:

```
while (TRUE) {
    while (TRUE) {
        cat("inner loop\n")
        break
    }
    cat("outer loop\n")
}
```


www.dbooks.org

With a little creativity, however, it is possible to break entirely from within a nested loop. As an example, consider the following expression, which, in its current state, will loop infinitely:

```
while (TRUE) {
    cat("outer loop body\n")
    while (TRUE) {
        cat("inner loop body\n")
        x <- runif(1)
        if (x < .3) {
            break
        } else {
            cat(sprintf("x is %.5f\n", x))
        }
    }
}
```

One possibility is to recognize that, unlike `break`, the `return` expression **does** have the ability to return control across multiple levels of enclosing loops. However, since `return` is only valid when used within a function, we cannot simply replace `break` with `return()` above, but also need to wrap the entire expression as an anonymous function:

```
(function() {
    while (TRUE) {
        cat("outer loop body\n")
        while (TRUE) {
            cat("inner loop body\n")
            x <- runif(1)
            if (x < .3) {
                return()
            } else {
                cat(sprintf("x is %.5f\n", x))
            }
        }
    }
})()
```

Alternatively, we can create a dummy variable (`exit`) prior to the expression, and activate it via `<<-` from the inner loop when we are ready to terminate:

```
exit <- FALSE
while (TRUE) {
    cat("outer loop body\n")
    while (TRUE) {
        cat("inner loop body\n")
        x <- runif(1)
        if (x < .3) {
            exit <<- TRUE
            break
        } else {
            cat(sprintf("x is %.5f\n", x))
        }
    }
    if (exit) break
}
```

Read Control flow structures online: https://riptutorial.com/r/topic/2201/control-flow-structures

Chapter 22: Creating packages with devtools

Introduction

This topic will cover the creation of R packages from scratch with the devtools package.

Remarks

1. Official R manual for creating packages
2. `roxygen2` reference manual
3. `devtools` reference manual

Examples

Creating and distributing packages

This is a *compact guide* about how to quickly create an R package from your code. Exhaustive documentations will be linked when available and should be read if you want a deeper knowledge of the situation. See *Remarks* for more resources.

The directory where your code stands will be refered as `./`, and all the commands are meant to be executed from a R prompt in this folder.

Creation of the documentation

The documentation for your code has to be in a format which is very similar to LaTeX.

However, we will use a tool named `roxygen` in order to simplify the process:

```
install.packages("devtools")
library("devtools")
install.packages("roxygen2")
library("roxygen2")
```

The full man page for roxygen is available here. It is very similar to *doxygen*.

Here is a practical sample about how to document a function with *roxygen*:

```
#' Increment a variable.
#'
#' Note that the behavior of this function
#' is undefined if `x` is not of class `numeric`.
#'
#' @export
#' @author  another guy
```


www.dbooks.org

```
#' @name    Increment Function
#' @title   increment
#'
#' @param x   Variable to increment
#' @return    `x` incremented of 1
#'
#' @seealso   `other_function`
#'
#' @examples
#' increment(3)
#' > 4
increment <- function(x) {
  return (x+1)
}
```

And here will be the result.

It is also recommanded to create a vignette (see the topic *Creating vignettes*), which is a full guide about your package.

Construction of the package skeleton

Assuming that your code is written for instance in files `./script1.R` and `./script2.R`, launch the following command in order to create the file tree of your package:

```
package.skeleton(name="MyPackage", code_files=c("script1.R","script2.R"))
```

Then delete all the files in `./MyPackage/man/`. You have now to compile the documentation:

```
roxygenize("MyPackage")
```

You should also generate a reference manual from your documentation using `R CMD Rd2pdf MyPackage` from a *command prompt* started in `./`.

Edition of the package properties

1. Package description

Modify `./MyPackage/DESCRIPTION` according to your needs. The fields `Package`, `Version`, `License`, `Description`, `Title`, `Author` and `Maintainer` are mandatory, the other are optional.

If your package depends on others packages, specify them in a field named `Depends` (R version < 3.2.0) or `Imports` (R version > 3.2.0).

2. Optional folders

Once you launched the skeleton build, `./MyPackage/` only had `R/` and `man/` subfolders. However, it can have some others:

- `data/`: here you can place the data that your library needs and that isn't code. It must be saved as dataset with the `.RData` extension, and you can load it at runtime with `data()` and `load()`
- `tests/`: all the code files in this folder will be ran at install time. If there is any error, the installation will fail.
- `src/`: for C/C++/Fortran source files you need (using `Rcpp`...).
- `exec/`: for other executables.
- `misc/`: for barely everything else.

Finalization and build

You can delete `./MyPackage/Read-and-delete-me`.

As it is now, your package is ready to be installed.

You can install it with `devtools::install("MyPackage")`.

To build your package as a source tarball, you need to execute the following command, from a *command prompt* in `./` : `R CMD build MyPackage`

Distribution of your package

Through Github

Simply create a new repository called *MyPackage* and upload everything in `MyPackage/` to the master branch. Here is an example.

Then anyone can install your package from github with devtools:

```
install_package("MyPackage", "your_github_usename")
```

Through CRAN

Your package needs to comply to the CRAN Repository Policy. Including but not limited to: your package must be cross-platforms (except some very special cases), it should pass the `R CMD check` test.

Here is the submission form. You must upload the source tarball.

Creating vignettes

www.dbooks.org

> A vignette is a long-form guide to your package. Function documentation is great if you know the name of the function you need, but it's useless otherwise. A vignette is like a book chapter or an academic paper: it can describe the problem that your package is designed to solve, and then show the reader how to solve it.

Vignettes will be created entirely in markdown.

Requirements

- Rmarkdown: `install.packages("rmarkdown")`
- Pandoc

Vignette creation

```
devtools::use_vignette("MyVignette", "MyPackage")
```

You can now edit your vignette at `./vignettes/MyVignette.Rmd`.

The text in your vignette is formatted as Markdown.

The only addition to the original Markdown, is a tag that takes R code, runs it, captures the output, and translates it into formatted Markdown:

````
```{r}
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
```
````

Will display as:

```
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
## [1] 30
```

Thus, all the packages you will use in your vignettes must be listed as dependencies in `./DESCRIPTION`.

Read Creating packages with devtools online: https://riptutorial.com/r/topic/10884/creating-packages-with-devtools

Chapter 23: Creating reports with RMarkdown

Examples

Printing tables

There are several packages that allow the output of data structures in form of HTML or LaTeX tables. They mostly differ in flexibility.

Here I use the packages:

- knitr
- xtable
- pander

For HTML documents

````
---
title: "Printing Tables"
author: "Martin Schmelzer"
date: "29 Juli 2016"
output: html_document
---

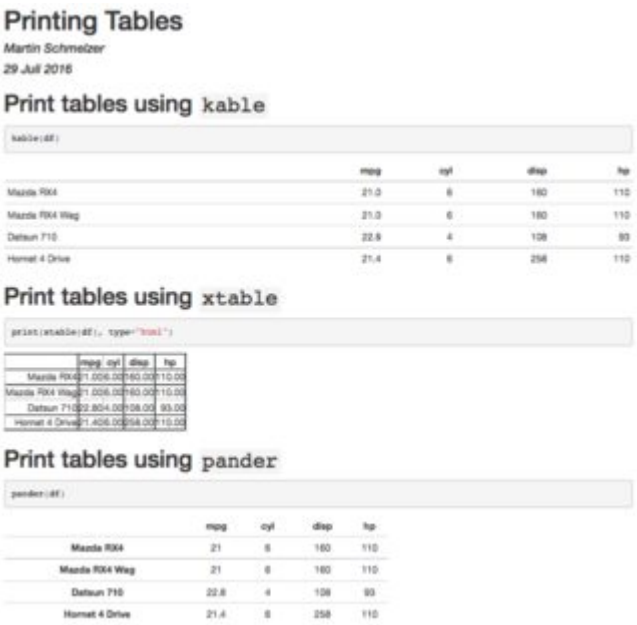
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
library(knitr)
library(xtable)
library(pander)
df <- mtcars[1:4,1:4]
```

# Print tables using `kable`
```{r, 'kable'}
kable(df)
```

# Print tables using `xtable`
```{r, 'xtable', results='asis'}
print(xtable(df), type="html")
```

# Print tables using `pander`
```{r, 'pander'}
pander(df)
```
````


www.dbooks.org



For PDF documents

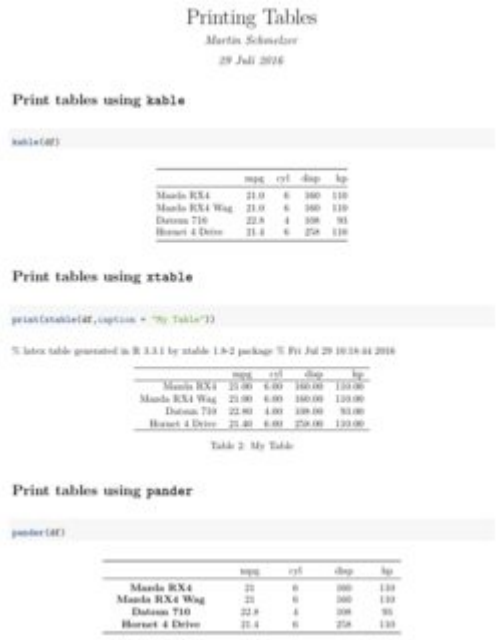
```
---
title: "Printing Tables"
author: "Martin Schmelzer"
date: "29 Juli 2016"
output: pdf_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
library(knitr)
library(xtable)
library(pander)
df <- mtcars[1:4,1:4]
```

Print tables using `kable`
```{r, 'kable'}
kable(df)
```

Print tables using `xtable`
```{r, 'xtable', results='asis'}
print(xtable(df, caption="My Table"))
```

Print tables using `pander`
```{r, 'pander'}
pander(df)
```
```



How can I stop xtable printing the comment ahead of each table?

```
options(xtable.comment = FALSE)
```

Including LaTeX Preample Commands

There are two possible ways of including LaTeX preamble commands (e.g. `\usepackage`) in a RMarkdown document.

1. Using the YAML option `header-includes`:

```
---
title: "Including LaTeX Preample Commands in RMarkdown"
header-includes:
   - \renewcommand{\familydefault}{cmss}
   - \usepackage[cm, slantedGreek]{sfmath}
   - \usepackage[T1]{fontenc}
output: pdf_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, external=T)
```

Section 1

As you can see, this text uses the Computer Moden Font!
```



www.dbooks.org

2. Including External Commands with `includes`, `in_header`

```
---
title: "Including LaTeX Preample Commands in RMarkdown"
output:
  pdf_document:
    includes:
      in_header: includes.tex
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, external=T)
```

Section 1

As you can see, this text uses the Computer Modern Font!
```

Here, the content of includes.tex are the same three commands we included with `header-includes`.

Writing a whole new template

A possible third option is to write your own LaTex template and include it with `template`. But this covers a lot more of the structure than only the preamble.

```
---
title: "My Template"
author: "Martin Schmelzer"
output:
  pdf_document:
    template: myTemplate.tex
---
```

Including bibliographies

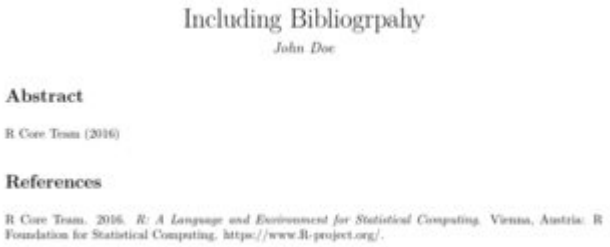
A bibtex catalogue cna easily be included with the YAML option `bibliography:`. A certain style for the bibliography can be added with `biblio-style:`. The references are added at the end of the document.

```
---
title: "Including Bibliography"
author: "John Doe"
output: pdf_document
bibliography: references.bib
---

# Abstract

@R_Core_Team_2016

# References
```



Including Bibliogrpahy

John Doe

Abstract

R Core Team (2016)

References

R Core Team. 2016. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Basic R-markdown document structure

R-markdown code chunks

R-markdown is a markdown file with embedded blocks of R code called *chunks*. There are two types of R code chunks: **inline** and **block**.

Inline chunks are added using the following syntax:

```
`r 2*2`
```

They are evaluated and inserted their output answer in place.

Block chunks have a different syntax:

````
```{r name, echo=TRUE, include=TRUE, ...}

2*2

````
````

And they come with several possible options. Here are the main ones (but there are many others):

- **echo** (boolean) controls wether the code inside chunk will be included in the document
- **include** (boolean) controls wether the output should be included in the document
- **fig.width** (numeric) sets the width of the output figures
- **fig.height** (numeric) sets the height of the output figures
- **fig.cap** (character) sets the figure captions

They are written in a simple `tag=value` format like in the example above.

R-markdown document example

Below is a basic example of R-markdown file illustrating the way R code chunks are embedded inside r-markdown.

```
# Title #

This is **plain markdown** text.
```


www.dbooks.org

```
```{r code, include=FALSE, echo=FALSE}

# Just declare variables

income <- 1000
taxes  <- 125

```

My income is: `r income ` dollars and I payed `r taxes ` dollars in taxes.

Below is the sum of money I will have left:

```{r gain, include=TRUE, echo=FALSE}

gain <- income-taxes

gain

```

```{r plotOutput, include=TRUE, echo=FALSE, fig.width=6, fig.height=6}

pie(c(income,taxes), label=c("income", "taxes"))

```
```

Converting R-markdown to other formats

The R `knitr` package can be used to evaluate R chunks inside R-markdown file and turn it into a regular markdown file.

The following steps are needed in order to turn R-markdown file into pdf/html:

1. Convert R-markdown file to markdown file using `knitr`.
2. Convert the obtained markdown file to pdf/html using specialized tools like *pandoc*.

In addition to the above `knitr` package has wrapper functions `knit2html()` and `knit2pdf()` that can be used to produce the final document without the intermediate step of manually converting it to the markdown format:

If the above example file was saved as `income.Rmd` it can be converted to a `pdf` file using the following R commands:

```
library(knitr)
knit2pdf("income.Rmd", "income.pdf")
```

The final document will be similar to the one below.

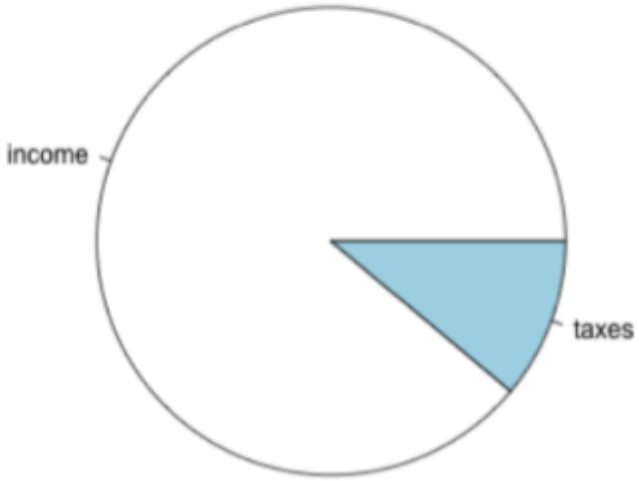
Title

This is **plain markdown** text.

My income is: 1000 dollars and I payed 125 dollars in taxes.

Below is the sum of money I will have left:

```
## [1] 875
```



Read Creating reports with RMarkdown online: https://riptutorial.com/r/topic/4572/creating-reports-with-rmarkdown

www.dbooks.org

Chapter 24: Creating vectors

Examples

Sequence of numbers

Use the `:` operator to create sequences of numbers, such as for use in vectorizing larger chunks of your code:

```
x <- 1:5
x
## [1] 1 2 3 4 5
```

This works both ways

```
10:4
# [1] 10  9  8  7  6  5  4
```

and even with floating point numbers

```
1.25:5
# [1] 1.25 2.25 3.25 4.25
```

or negatives

```
-4:4
#[1] -4 -3 -2 -1  0  1  2  3  4
```

seq()

`seq` is a more flexible function than the `:` operator allowing to specify steps other than 1.

The function creates a sequence from the `start` (default is 1) to the end including that number.

You can supply only the end (`to`) parameter

```
seq(5)
# [1] 1 2 3 4 5
```

As well as the start

```
seq(2, 5) # or seq(from=2, to=5)
# [1] 2 3 4 5
```

And finally the step (`by`)

```
seq(2, 5, 0.5) # or seq(from=2, to=5, by=0.5)
# [1] 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

`seq` can optionally infer the (evenly spaced) steps when alternatively the desired length of the output (`length.out`) is supplied

```
seq(2,5, length.out = 10)
# [1] 2.0 2.3 2.6 2.9 3.2 3.5 3.8 4.1 4.4 4.7 5.0
```

If the sequence needs to have the same length as another vector we can use the `along.with` as a shorthand for `length.out = length(x)`

```
x = 1:8
seq(2,5,along.with = x)
# [1] 2.000000 2.428571 2.857143 3.285714 3.714286 4.142857 4.571429 5.000000
```

There are two useful simplified functions in the `seq` family: `seq_along`, `seq_len`, and `seq.int`. `seq_along` and `seq_len` functions construct the natural (counting) numbers from 1 through N where N is determined by the function argument, the length of a vector or list with `seq_along`, and the integer argument with `seq_len`.

```
seq_along(x)
# [1] 1 2 3 4 5 6 7 8
```

Note that `seq_along` returns the indices of an existing object.

```
# counting numbers 1 through 10
seq_len(10)
[1]  1  2  3  4  5  6  7  8  9 10
# indices of existing vector (or list) with seq_along
letters[1:10]
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
seq_along(letters[1:10])
[1]  1  2  3  4  5  6  7  8  9 10
```

`seq.int`is the same as `seq` maintained for ancient compatibility.

There is also an old function `sequence`that creates a vector of sequences from a non negative argument.

```
sequence(4)
# [1] 1 2 3 4
sequence(c(3, 2))
# [1] 1 2 3 1 2
sequence(c(3, 2, 5))
# [1] 1 2 3 1 2 1 2 3 4 5
```

Vectors

Vectors in R can have different types (e.g. integer, logical, character). The most general way of

www.dbooks.org

defining a vector is by using the function `vector()`.

```
vector('integer',2) # creates a vector of integers of size 2.
vector('character',2) # creates a vector of characters of size 2.
vector('logical',2) # creates a vector of logicals of size 2.
```

However, in R, the shorthand functions are generally more popular.

```
integer(2) # is the same as vector('integer',2) and creates an integer vector with two
elements
character(2) # is the same as vector('integer',2) and creates an character vector with two
elements
logical(2) # is the same as vector('logical',2) and creates an logical vector with two
elements
```

Creating vectors with values, other than the default values, is also possible. Often the function `c()` is used for this. The c is short for combine or concatenate.

```
c(1, 2) # creates a integer vector of two elements: 1 and 2.
c('a', 'b') # creates a character vector of two elements: a and b.
c(T,F) # creates a logical vector of two elements: TRUE and FALSE.
```

Important to note here is that R interprets any integer (e.g. 1) as an integer vector of size one. The same holds for numerics (e.g. 1.1), logicals (e.g. T or F), or characters (e.g. 'a'). Therefore, you are in essence combining vectors, which in turn are vectors.

Pay attention that you always have to combine similar vectors. Otherwise, R will try to convert the vectors in vectors of the same type.

```
c(1,1.1,'a',T) # all types (integer, numeric, character and logical) are converted to the
'lowest' type which is character.
```

Finding elements in vectors can be done with the `[` operator.

```
vec_int <- c(1,2,3)
vec_char <- c('a','b','c')
vec_int[2] # accessing the second element will return 2
vec_char[2] # accessing the second element will return 'b'
```

This can also be used to change values

```
vec_int[2] <- 5 # change the second value from 2 to 5
vec_int # returns [1] 1 5 3
```

Finally, the `:` operator (short for the function `seq()`) can be used to quickly create a vector of numbers.

```
vec_int <- 1:10
vec_int # returns [1] 1 2 3 4 5 6 7 8 9 10
```

This can also be used to subset vectors (from easy to more complex subsets)

```
vec_char <- c('a','b','c','d','e')
vec_char[2:4] # returns [1] "b" "c" "d"
vec_char[c(1,3,5)] # returns [1] "a" "c" "e"
```

Creating named vectors

Named vector can be created in several ways. With `c`:

```
xc <- c('a' = 5, 'b' = 6, 'c' = 7, 'd' = 8)
```

which results in:

```
> xc
a b c d
5 6 7 8
```

with `list`:

```
xl <- list('a' = 5, 'b' = 6, 'c' = 7, 'd' = 8)
```

which results in:

```
> xl
$a
[1] 5

$b
[1] 6

$c
[1] 7

$d
[1] 8
```

With the `setNames` function, two vectors of the same length can be used to create a named vector:

```
x <- 5:8
y <- letters[1:4]

xy <- setNames(x, y)
```

which results in a named integer vector:

```
> xy
a b c d
5 6 7 8
```

As can be seen, this gives the same result as the `c` method.

www.dbooks.org

You may also use the `names` function to get the same result:

```
xy <- 5:8
names(xy) <- letters[1:4]
```

With such a vector it is also possibly to select elements by name:

```
> xy["c"]
c
7
```

This feature makes it possible to use such a named vector as a look-up vector/table to match the values to values of another vector or column in dataframe. Considering the following dataframe:

```
mydf <- data.frame(let = c('c','a','b','d'))

> mydf
  let
1   c
2   a
3   b
4   d
```

Suppose you want to create a new variable in the `mydf` dataframe called `num` with the correct values from `xy` in the rows. Using the `match` function the appropriate values from `xy` can be selected:

```
mydf$num <- xy[match(mydf$let, names(xy))]
```

which results in:

```
> mydf
  let num
1   c   7
2   a   5
3   b   6
4   d   8
```

Expanding a vector with the rep() function

The `rep` function can be used to repeat a vector in a fairly flexible manner.

```
# repeat counting numbers, 1 through 5 twice
rep(1:5, 2)
[1] 1 2 3 4 5 1 2 3 4 5

# repeat vector with incomplete recycling
rep(1:5, 2, length.out=7)
[1] 1 2 3 4 5 1 2
```

The each argument is especially useful for expanding a vector of statistics of observational/experimental units into a vector of data.frame with repeated observations of these

units.

```
# same except repeat each integer next to each other
rep(1:5, each=2)
[1] 1 1 2 2 3 3 4 4 5 5
```

A nice feature of `rep` regarding involving expansion to such a data structure is that expansion of a vector to an unbalanced panel can be accomplished by replacing the length argument with a vector that dictates the number of times to repeat each element in the vector:

```
# automated length repetition
rep(1:5, 1:5)
 [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
# hand-fed repetition length vector
rep(1:5, c(1,1,1,2,2))
[1] 1 2 3 4 4 5 5
```

This should expose the possibility of allowing an external function to feed the second argument of `rep` in order to dynamically construct a vector that expands according to the data.

As with `seq`, faster, simplified versions of `rep` are `rep_len` and `rep.int`. These drop some attributes that `rep` maintains and so may be most useful in situations where speed is a concern and additional aspects of the repeated vector are unnecessary.

```
# repeat counting numbers, 1 through 5 twice
rep.int(1:5, 2)
[1] 1 2 3 4 5 1 2 3 4 5

# repeat vector with incomplete recycling
rep_len(1:5, length.out=7)
[1] 1 2 3 4 5 1 2
```

Vectors from build in constants: Sequences of letters & month names

`R` has a number of build in constants. The following constants are available:

- `LETTERS`: the 26 upper-case letters of the Roman alphabet
- `letters`: the 26 lower-case letters of the Roman alphabet
- `month.abb`: the three-letter abbreviations for the English month names
- `month.name`: the English names for the months of the year
- `pi`: the ratio of the circumference of a circle to its diameter

From the letters and month constants, vectors can be created.

1) Sequences of letters:

```
> letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
"w" "x" "y" "z"
```


www.dbooks.org

```
> LETTERS[7:9]
[1] "G" "H" "I"

> letters[c(1,5,3,2,4)]
[1] "a" "e" "c" "b" "d"
```

2) Sequences of month abbreviations or month names:

```
> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

> month.name[1:4]
[1] "January"  "February" "March"    "April"

> month.abb[c(3,6,9,12)]
[1] "Mar" "Jun" "Sep" "Dec"
```

Read Creating vectors online: https://riptutorial.com/r/topic/1088/creating-vectors

Chapter 25: Data acquisition

Introduction

Get data directly into an R session. One of the nice features of R is the ease of data acquisition. There are several ways data dissemination using R packages.

Examples

Built-in datasets

Rhas a vast collection of built-in datasets. Usually, they are used for teaching purposes to create quick and easily reproducible examples. There is a nice web-page listing the built-in datasets:

https://vincentarelbundock.github.io/Rdatasets/datasets.html

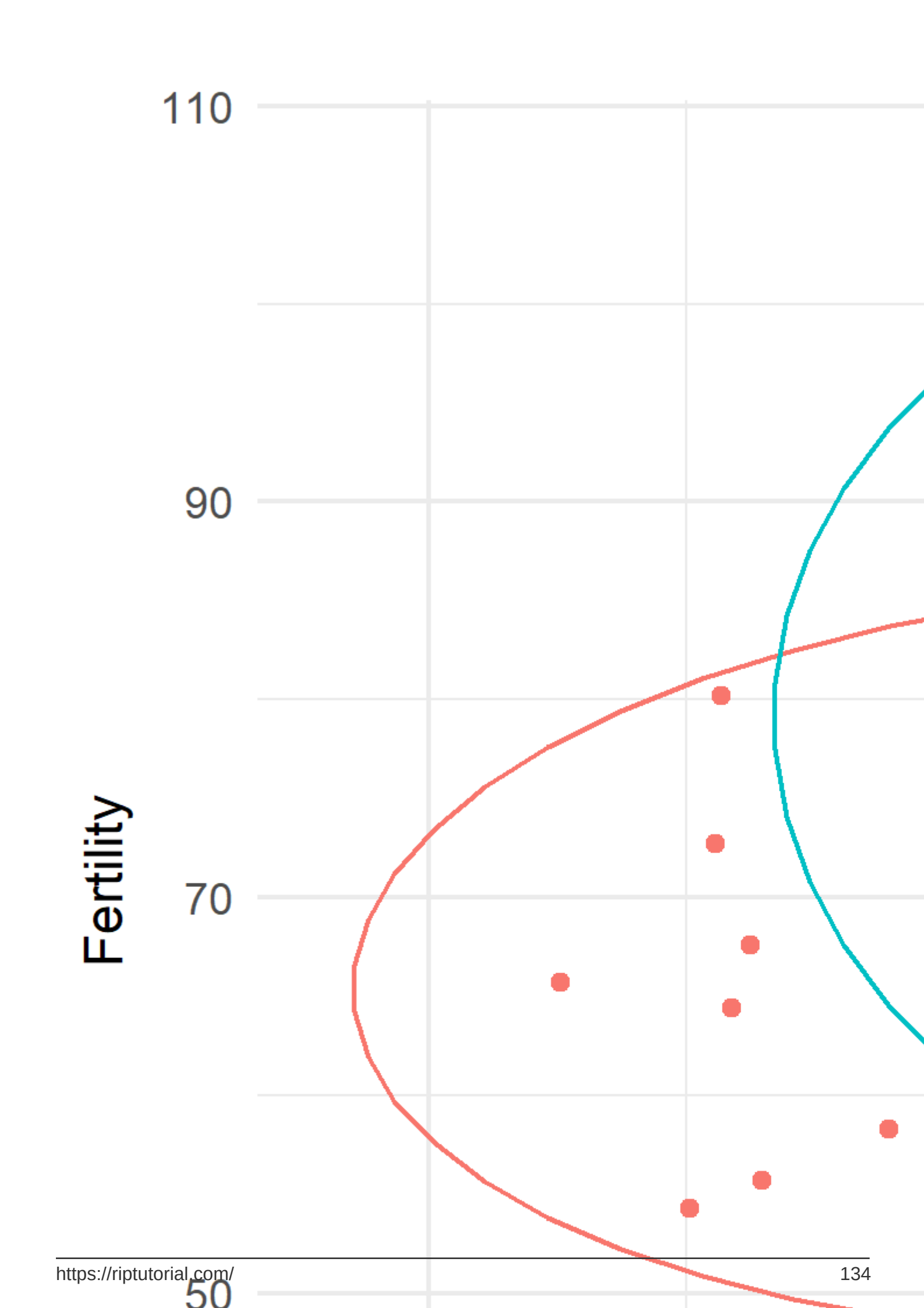
Example

Swiss Fertility and Socioeconomic Indicators (1888) Data. Let's check the difference in fertility based of rurality and domination of Catholic population.

```
library(tidyverse)

swiss %>%
        ggplot(aes(x = Agriculture, y = Fertility,
                   color = Catholic > 50))+
        geom_point()+
        stat_ellipse()
```


www.dbooks.org


110
90
70
50
Fertility

, it does not find all the relevant datasets available. This, it's more convenient to browse the code of a dataset manually at the Eurostat website: Countries Database, or Regional Database. If the automated download does not work, the data can be grabbed manually at via Bulk Download Facility.

```
library(tidyverse)
library(lubridate)
library(forcats)
library(eurostat)
library(geofacet)
library(viridis)
library(ggthemes)
library(extrafont)

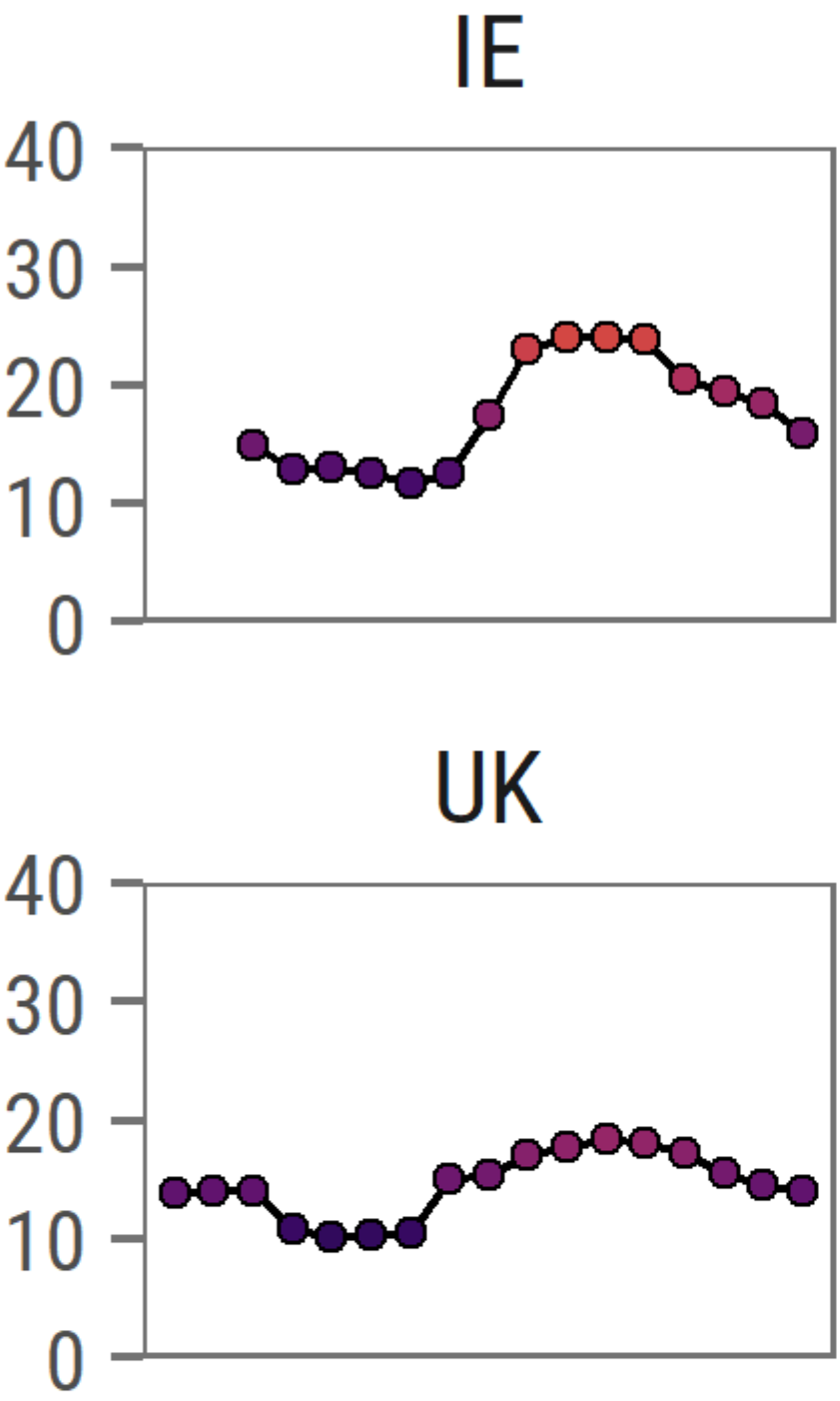
# download NEET data for countries
neet <- get_eurostat("edat_lfse_22")

neet %>%
        filter(geo %>% paste %>% nchar == 2,
               sex == "T", age == "Y18-24") %>%
        group_by(geo) %>%
        mutate(avg = values %>% mean()) %>%
        ungroup() %>%
        ggplot(aes(x = time %>% year(),
                   y = values))+
        geom_path(aes(group = 1))+
        geom_point(aes(fill = values), pch = 21)+
        scale_x_continuous(breaks = seq(2000, 2015, 5),
                           labels = c("2000", "'05", "'10", "'15"))+
        scale_y_continuous(expand = c(0, 0), limits = c(0, 40))+
        scale_fill_viridis("NEET, %", option = "B")+
        facet_geo(~ geo, grid = "eu_grid1")+
        labs(x = "Year",
             y = "NEET, %",
             title = "Young people neither in employment nor in education and training in
Europe",
             subtitle = "Data: Eurostat Regional Database, 2000-2016",
             caption = "ikashnitsky.github.io")+
        theme_few(base_family =  "Roboto Condensed", base_size = 15)+
        theme(axis.text = element_text(size = 10),
              panel.spacing.x = unit(1, "lines"),
              legend.position = c(0, 0),
              legend.justification = c(0, 0))
```


www.dbooks.org

Young people neith

Data: Eurostat Regional D



that gathers and pre-process human mortality data for those countries, where more or less reliable statistics is available.

```
# load required packages
library(tidyverse)
library(extrafont)
library(HMDHFDplus)

country <- getHMDcountries()

exposures <- list()
for (i in 1: length(country)) {
        cnt <- country[i]
        exposures[[cnt]] <- readHMDweb(cnt, "Exposures_1x1", user_hmd, pass_hmd)
        # let's print the progress
        paste(i,'out of',length(country))
} # this will take quite a lot of time
```

Please note, the arguments `user_hmd` and `pass_hmd` are the login credentials at the website of Human Mortality Database. In order to access the data, one needs to create an account at http://www.mortality.org/ and provide their own credentials to the `readHMDweb()` function.

```
sr_age <- list()

for (i in 1:length(exposures)) {
        di <- exposures[[i]]
        sr_agei <- di %>% select(Year,Age,Female,Male) %>%
                filter(Year %in% 2012) %>%
                select(-Year) %>%
                transmute(country = names(exposures)[i],
                          age = Age, sr_age = Male / Female * 100)
        sr_age[[i]] <- sr_agei
}
sr_age <- bind_rows(sr_age)

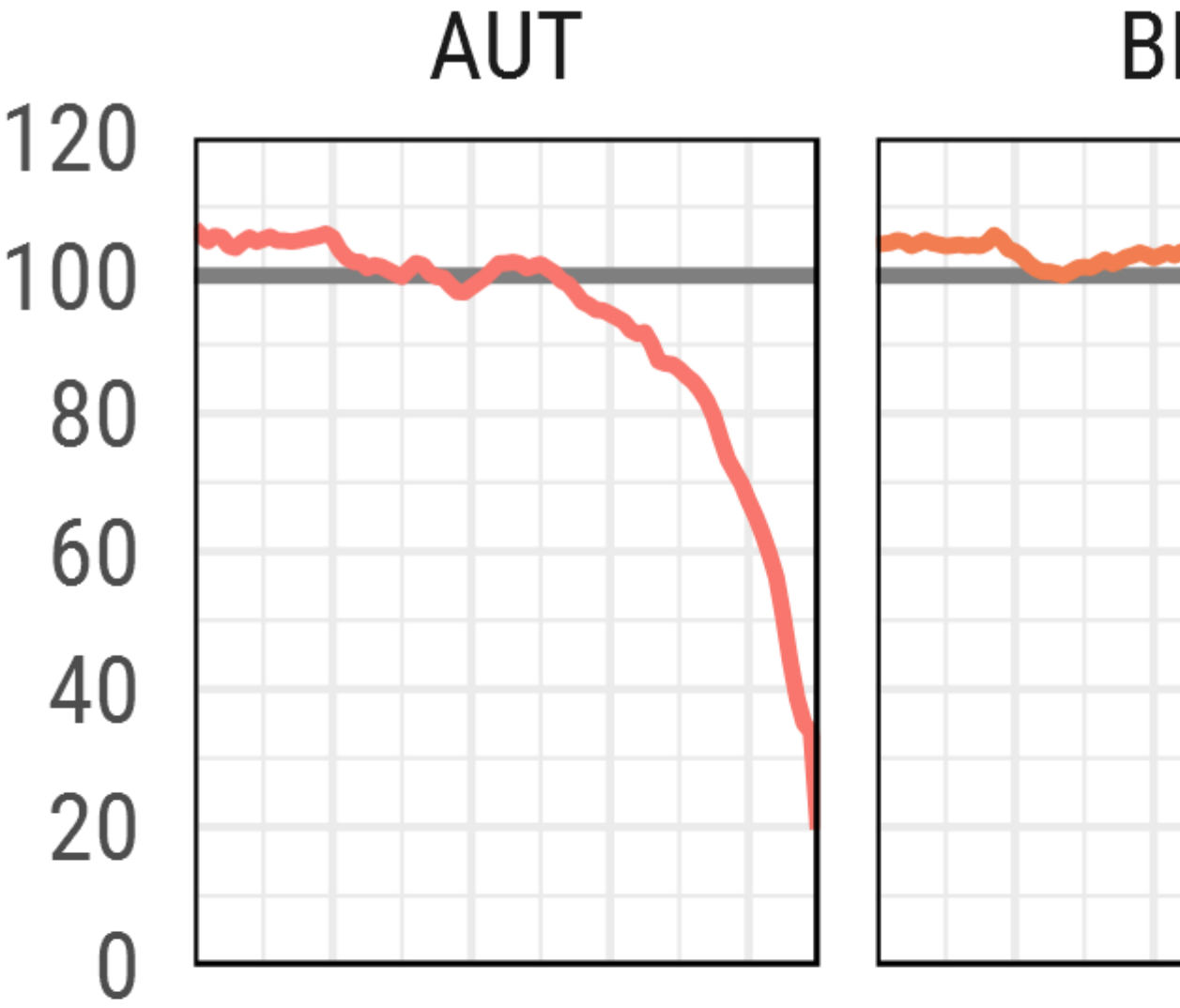
# remove optional populations
sr_age <- sr_age %>% filter(!country %in% c("FRACNP","DEUTE","DEUTW","GBRCENW","GBR_NP"))

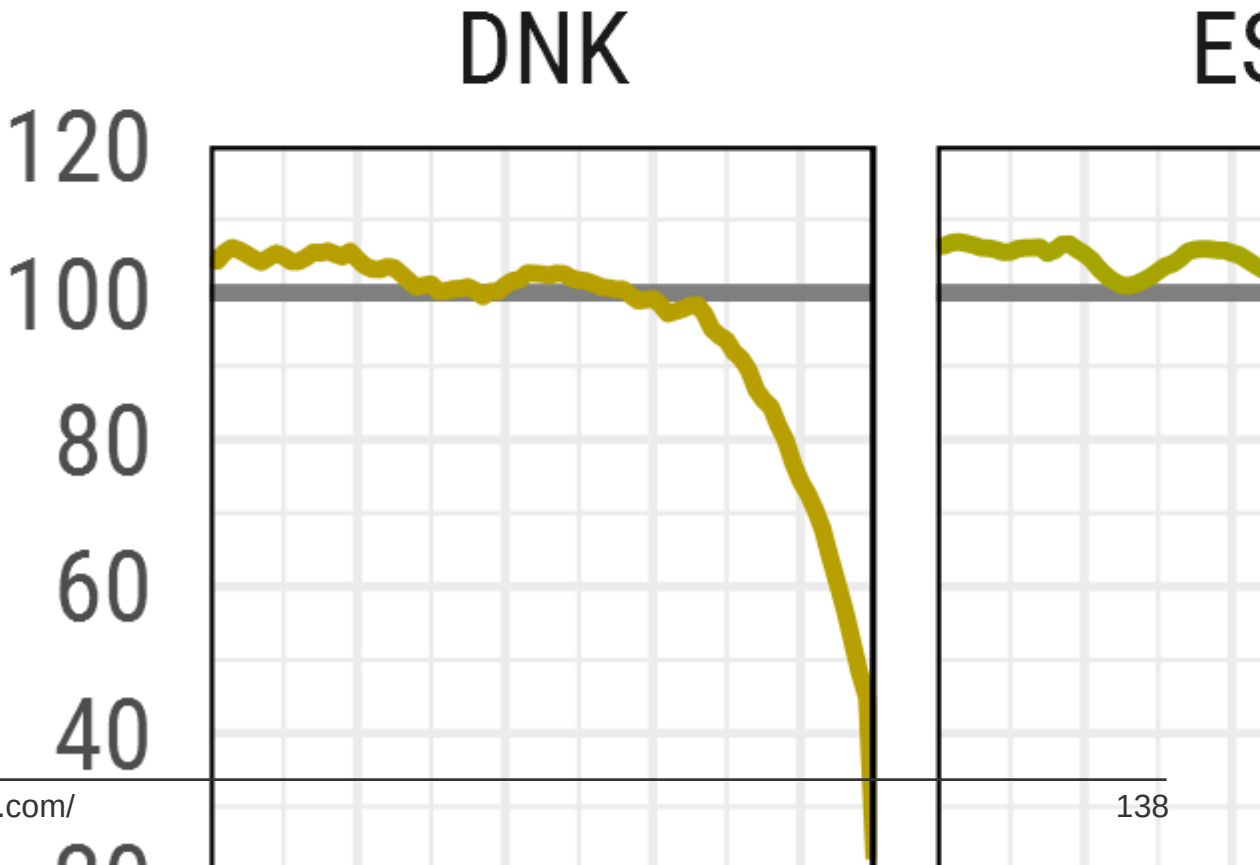
# summarize all ages older than 90 (too jerky)
sr_age_90 <- sr_age %>% filter(age %in% 90:110) %>%
        group_by(country) %>% summarise(sr_age = mean(sr_age, na.rm = T)) %>%
        ungroup() %>% transmute(country, age=90, sr_age)

df_plot <- bind_rows(sr_age %>% filter(!age %in% 90:110), sr_age_90)

# finaly - plot
df_plot %>%
        ggplot(aes(age, sr_age, color = country, group = country))+
        geom_hline(yintercept = 100, color = 'grey50', size = 1)+
        geom_line(size = 1)+
        scale_y_continuous(limits = c(0, 120), expand = c(0, 0), breaks = seq(0, 120, 20))+
        scale_x_continuous(limits = c(0, 90), expand = c(0, 0), breaks = seq(0, 80, 20))+
        xlab('Age')+
        ylab('Sex ratio, males per 100 females')+
        facet_wrap(~country, ncol=6)+
        theme_minimal(base_family = "Roboto Condensed", base_size = 15)+
        theme(legend.position='none',
              panel.border = element_rect(size = .5, fill = NA))
```


www.dbooks.org


AUT
120
100
80
60
40
20
0


DNK
120
100
80
60
40

Chapter 26: Data frames

Syntax

- data.frame(..., row.names = NULL, check.rows = FALSE, check.names = TRUE, stringsAsFactors = default.stringsAsFactors())
- as.data.frame(x, row.names = NULL, optional = FALSE, ...) # generic function
- as.data.frame(x, ..., stringsAsFactors = default.stringsAsFactors()) # S3 method for class 'character'
- as.data.frame(x, row.names = NULL, optional = FALSE, ..., stringsAsFactors = default.stringsAsFactors()) # S3 method for class 'matrix'
- is.data.frame(x)

Examples

Create an empty data.frame

A data.frame is a special kind of list: it is *rectangular*. Each element (column) of the list has same length, and where each row has a "row name". Each column has its own class, but the class of one column can be different from the class of another column (unlike a matrix, where all elements must have the same class).

In principle, a data.frame could have no rows and no columns:

```
> structure(list(character()), class = "data.frame")
NULL
<0 rows> (or 0-length row.names)
```

But this is unusual. It is more common for a data.frame to have many columns and many rows. Here is a data.frame with three rows and two columns (`a` is numeric class and `b` is character class):

```
> structure(list(a = 1:3, b = letters[1:3]), class = "data.frame")
[1] a b
<0 rows> (or 0-length row.names)
```

In order for the data.frame to print, we will need to supply some row names. Here we use just the numbers 1:3:

```
> structure(list(a = 1:3, b = letters[1:3]), class = "data.frame", row.names = 1:3)
  a b
1 1 a
2 2 b
3 3 c
```


www.dbooks.org

Now it becomes obvious that we have a data.frame with 3 rows and 2 columns. You can check this using `nrow()`, `ncol()`, and `dim()`:

```
> x <- structure(list(a = numeric(3), b = character(3)), class = "data.frame", row.names =
1:3)
> nrow(x)
[1] 3
> ncol(x)
[1] 2
> dim(x)
[1] 3 2
```

R provides two other functions (besides `structure()`) that can be used to create a data.frame. The first is called, intuitively, `data.frame()`. It checks to make sure that the column names you supplied are valid, that the list elements are all the same length, and supplies some automatically generated row names. This means that the output of `data.frame()` might now always be exactly what you expect:

```
> str(data.frame("a a a" = numeric(3), "b-b-b" = character(3)))
'data.frame':   3 obs. of  2 variables:
 $ a.a.a: num  0 0 0
 $ b.b.b: Factor w/ 1 level "": 1 1 1
```

The other function is called `as.data.frame()`. This can be used to coerce an object that is not a data.frame into being a data.frame by running it through `data.frame()`. As an example, consider a matrix:

```
> m <- matrix(letters[1:9], nrow = 3)
> m
     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"
```

And the result:

```
> as.data.frame(m)
  V1 V2 V3
1  a  d  g
2  b  e  h
3  c  f  i
> str(as.data.frame(m))
'data.frame':   3 obs. of  3 variables:
 $ V1: Factor w/ 3 levels "a","b","c": 1 2 3
 $ V2: Factor w/ 3 levels "d","e","f": 1 2 3
 $ V3: Factor w/ 3 levels "g","h","i": 1 2 3
```

Subsetting rows and columns from a data frame

Syntax for accessing rows and columns: `[,` `[[,`

and `$`

This topic covers the most common syntax to access specific rows and columns of a data frame. These are

- Like a `matrix` with single brackets `data[rows, columns]`
  - Using row and column numbers
  - Using column (and row) names
- Like a `list`:
  - With single brackets `data[columns]` to get a data frame
  - With double brackets `data[[one_column]]` to get a vector
- With `$` for a single column `data$column_name`

We will use the built-in `mtcars` data frame to illustrate.

Like a matrix: `data[rows, columns]`

With numeric indexes

Using the built in data frame `mtcars`, we can extract rows and columns using `[]` brackets with a comma included. Indices before the comma are rows:

```
# get the first row
mtcars[1, ]
# get the first five rows
mtcars[1:5, ]
```

Similarly, after the comma are columns:

```
# get the first column
mtcars[, 1]
# get the first, third and fifth columns:
mtcars[, c(1, 3, 5)]
```

As shown above, if either rows or columns are left blank, all will be selected. `mtcars[1, ]` indicates the first row with *all* the columns.

With column (and row) names

So far, this is identical to how rows and columns of matrices are accessed. With `data.frame`s, most of the time it is preferable to use a column name to a column index. This is done by using a `character` with the column name instead of `numeric` with a column number:

```
# get the mpg column
mtcars[, "mpg"]
# get the mpg, cyl, and disp columns
mtcars[, c("mpg", "cyl", "disp")]
```


www.dbooks.org

Though less common, row names can also be used:

```
mtcars["Mazda Rx4", ]
```

Rows and columns together

The row and column arguments can be used together:

```
# first four rows of the mpg column
mtcars[1:4, "mpg"]

# 2nd and 5th row of the mpg, cyl, and disp columns
mtcars[c(2, 5), c("mpg", "cyl", "disp")]
```

A warning about dimensions:

When using these methods, if you extract multiple columns, you will get a data frame back. However, if you extract a *single* column, you will get a vector, not a data frame under the default options.

```
## multiple columns returns a data frame
class(mtcars[, c("mpg", "cyl")])
# [1] "data.frame"
## single column returns a vector
class(mtcars[, "mpg"])
# [1] "numeric"
```

There are two ways around this. One is to treat the data frame as a list (see below), the other is to add a `drop = FALSE` argument. This tells R to not "drop the unused dimensions":

```
class(mtcars[, "mpg", drop = FALSE])
# [1] "data.frame"
```

Note that matrices work the same way - by default a single column or row will be a vector, but if you specify `drop = FALSE` you can keep it as a one-column or one-row matrix.

Like a list

Data frames are essentially `list`s, i.e., they are a list of column vectors (that all must have the same length). Lists can be subset using single brackets `[` for a sub-list, or double brackets `[[` for a single element.

With single brackets `data[columns]`

When you use single brackets and no commas, you will get column back because data frames are lists of columns.

```
mtcars["mpg"]
```

```
mtcars[c("mpg", "cyl", "disp")]
my_columns <- c("mpg", "cyl", "hp")
mtcars[my_columns]
```

Single brackets *like a list* vs. single brackets *like a matrix*

The difference between `data[columns]` and `data[, columns]` is that when treating the `data.frame` as a `list` (no comma in the brackets) the object returned will *be a* `data.frame`. If you use a comma to treat the `data.frame` like a `matrix` then selecting a single column will return a vector but selecting multiple columns will return a `data.frame`.

```
## When selecting a single column
## like a list will return a data frame
class(mtcars["mpg"])
# [1] "data.frame"
## like a matrix will return a vector
class(mtcars[, "mpg"])
# [1] "numeric"
```

With double brackets `data[[one_column]]`

To extract a single column *as a vector* when treating your `data.frame` as a `list`, you can use double brackets `[[`. This will only work for a single column at a time.

```
# extract a single column by name as a vector
mtcars[["mpg"]]

# extract a single column by name as a data frame (as above)
mtcars["mpg"]
```

Using `$` to access columns

A single column can be extracted using the magical shortcut `$` without using a quoted column name:

```
# get the column "mpg"
mtcars$mpg
```

Columns accessed by `$` will always be vectors, not data frames.

Drawbacks of `$` for accessing columns

The `$` can be a convenient shortcut, especially if you are working in an environment (such as RStudio) that will auto-complete the column name in this case. **However,** `$` has drawbacks as well: it uses *non-standard evaluation* to avoid the need for quotes, which means it *will not work* if your column name is stored in a variable.

```
my_column <- "mpg"
# the below will not work
mtcars$my_column
```


www.dbooks.org

```
# but these will work
mtcars[, my_column]  # vector
mtcars[my_column]    # one-column data frame
mtcars[[my_column]]  # vector
```

Due to these concerns, `$` is best used in *interactive* R sessions when your column names are constant. For *programmatic* use, for example in writing a generalizable function that will be used on different data sets with different column names, `$` should be avoided.

Also note that the default behaviour is to use partial matching only when extracting from recursive objects (except environments) by `$`

```
# give you the values of "mpg" column
# as "mtcars" has only one column having name starting with "m"
mtcars$m
# will give you "NULL"
# as "mtcars" has more than one columns having name starting with "d"
mtcars$d
```

Advanced indexing: negative and logical indices

Whenever we have the option to use numbers for a index, we can also use negative numbers to omit certain indices or a boolean (logical) vector to indicate exactly which items to keep.

Negative indices omit elements

```
mtcars[1, ]   # first row
mtcars[ -1, ] # everything but the first row
mtcars[-(1:10), ] # everything except the first 10 rows
```

Logical vectors indicate specific elements to keep

We can use a condition such as `<` to generate a logical vector, and extract only the rows that meet the condition:

```
# logical vector indicating TRUE when a row has mpg less than 15
# FALSE when a row has mpg >= 15
test <- mtcars$mpg < 15

# extract these rows from the data frame
mtcars[test, ]
```

We can also bypass the step of saving the intermediate variable

```
# extract all columns for rows where the value of cyl is 4.
mtcars[mtcars$cyl == 4, ]
```

```
# extract the cyl, mpg, and hp columns where the value of cyl is 4
mtcars[mtcars$cyl == 4, c("cyl", "mpg", "hp")]
```

Convenience functions to manipulate data.frames

Some convenience functions to manipulate `data.frames` are `subset()`, `transform()`, `with()` and `within()`.

subset

The `subset()` function allows you to subset a `data.frame` in a more convenient way (subset also works with other classes):

```
subset(mtcars, subset = cyl == 6, select = c("mpg", "hp"))
                mpg  hp
Mazda RX4      21.0 110
Mazda RX4 Wag  21.0 110
Hornet 4 Drive 21.4 110
Valiant        18.1 105
Merc 280       19.2 123
Merc 280C      17.8 123
Ferrari Dino   19.7 175
```

In the code above we asking only for the lines in which `cyl == 6` and for the columns `mpg` and `hp`. You could achieve the same result using `[]` with the following code:

```
mtcars[mtcars$cyl == 6, c("mpg", "hp")]
```

transform

The `transform()` function is a convenience function to change columns inside a `data.frame`. For instance the following code adds another column named `mpg2` with the result of `mpg^2` to the `mtcars` `data.frame`:

```
mtcars <- transform(mtcars, mpg2 = mpg^2)
```

with and within

Both `with()` and `within()` let you to evaluate expressions inside the `data.frame` environment, allowing a somewhat cleaner syntax, saving you the use of some `$` or `[]`.

For example, if you want to create, change and/or remove multiple columns in the `airquality` `data.frame`:

```
aq <- within(airquality, {
    lOzone <- log(Ozone) # creates new column
    Month <- factor(month.abb[Month]) # changes Month Column
    cTemp <- round((Temp - 32) * 5/9, 1) # creates new column
```


www.dbooks.org

```
    S.cT <- Solar.R / cTemp  # creates new column
    rm(Day, Temp) # removes columns
})
```

Introduction

Data frames are likely the data structure you will used most in your analyses. A data frame is a special kind of list that stores same-length vectors of different classes. You create data frames using the `data.frame` function. The example below shows this by combining a numeric and a character vector into a data frame. It uses the `:` operator, which will create a vector containing all integers from 1 to 3.

```
df1 <- data.frame(x = 1:3, y = c("a", "b", "c"))
df1
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
class(df1)
## [1] "data.frame"
```

Data frame objects do not print with quotation marks, so the class of the columns is not always obvious.

```
df2 <- data.frame(x = c("1", "2", "3"), y = c("a", "b", "c"))
df2
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
```

Without further investigation, the "x" columns in `df1` and `df2` cannot be differentiated. The `str` function can be used to describe objects with more detail than class.

```
str(df1)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: int  1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
str(df2)
## 'data.frame':    3 obs. of  2 variables:
##  $ x: Factor w/ 3 levels "1","2","3": 1 2 3
##  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Here you see that `df1` is a `data.frame` and has 3 observations of 2 variables, "x" and "y." Then you are told that "x" has the data type integer (not important for this class, but for our purposes it behaves like a numeric) and "y" is a factor with three levels (another data class we are not discussing). ***It is important to note that, by default, data frames coerce characters to factors.*** The default behavior can be changed with the `stringsAsFactors` parameter:

```
df3 <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)
str(df3)
## 'data.frame':    3 obs. of  2 variables:
```

```
##  $ x: int  1 2 3
##  $ y: chr  "a" "b" "c"
```

Now the "y" column is a character. As mentioned above, each "column" of a data frame must have the same length. Trying to create a data.frame from vectors with different lengths will result in an error. (Try running `data.frame(x = 1:3, y = 1:4)` to see the resulting error.)

As test-cases for data frames, some data is provided by R by default. One of them is iris, loaded as follows:

```
mydataframe <- iris
str(mydataframe)
```

Convert data stored in a list to a single data frame using do.call

If you have your data stored in a list and you want to convert this list to a data frame the `do.call` function is an easy way to achieve this. However, it is important that all list elements have the same length in order to prevent unintended recycling of values.

```
dataList  <- list(1:3,4:6,7:9)
dataList
# [[1]]
# [1] 1 2 3
#
# [[2]]
# [1] 4 5 6
#
# [[3]]
# [1] 7 8 9

dataframe <- data.frame(do.call(rbind, dataList))
dataframe
#   X1 X2 X3
# 1  1  2  3
# 2  4  5  6
# 3  7  8  9
```

It also works if your list consists of data frames itself.

```
dataframeList  <- list(data.frame(a = 1:2, b = 1:2, c = 1:2),
                       data.frame(a = 3:4, b = 3:4, c = 3:4))
dataframeList
# [[1]]
#   a b c
# 1 1 1 1
# 2 2 2 2

# [[2]]
#   a b c
# 1 3 3 3
# 2 4 4 4

dataframe      <- do.call(rbind, dataframeList)
dataframe
```


www.dbooks.org

```
#   a b c
# 1 1 1 1
# 2 2 2 2
# 3 3 3 3
# 4 4 4 4
```

Convert all columns of a data.frame to character class

A common task is to convert all columns of a data.frame to character class for ease of manipulation, such as in the cases of sending data.frames to a RDBMS or merging data.frames containing factors where levels may differ between input data.frames.

The best time to do this is when the data is read in - almost all input methods that create data frames have an options `stringsAsFactors` which can be set to `FALSE`.

If the data has already been created, factor columns can be converted to character columns as shown below.

```
bob <- data.frame(jobs = c("scientist", "analyst"),
                  pay  = c(160000, 100000), age = c(30, 25))
str(bob)
```

```
'data.frame':    2 obs. of  3 variables:
 $ jobs: Factor w/ 2 levels "analyst","scientist": 2 1
 $ pay : num  160000 100000
 $ age : num  30 25
```

```
# Convert *all columns* to character
bob[] <- lapply(bob, as.character)
str(bob)
```

```
    'data.frame':    2 obs. of  3 variables:
 $ jobs: chr  "scientist" "analyst"
 $ pay : chr  "160000" "1e+05"
 $ age : chr  "30" "25"
```

```
# Convert only factor columns to character
bob[] <- lapply(bob, function(x) {
    if is.factor(x) x <- as.character(x)
    return(x)
})
```

Subsetting Rows by Column Values

Built in functions can subset `rows` with `columns` that meet conditions.

```
df <- data.frame(item = c(1:10),
                 price_Elasticity = c(-0.57667, 0.03205, -0.04904, 0.10342, 0.04029,
                                      0.0742, 0.1669, 0.0313, 0.22204, 0.06158),
                 total_Margin = c(-145062, 98671, 20576, -56382, 207623, 43463, 1235,
                                  34521, 146553, -74516))
```

To find `rows` with `price_Elasticity > 0`:

```
df[df$price_Elasticity > 0, ]
```

```
   item price_Elasticity total_Margin
2     2          0.03205        98671
4     4          0.10342       -56382
5     5          0.04029       207623
6     6          0.07420        43463
7     7          0.16690         1235
8     8          0.03130        34521
9     9          0.22204       146553
10   10          0.06158       -74516
```

subset based on `price_Elasticity > 0` and `total_Margin > 0`:

```
df[df$price_Elasticity > 0 & df$total_Margin > 0, ]
```

```
  item price_Elasticity total_Margin
2    2          0.03205        98671
5    5          0.04029       207623
6    6          0.07420        43463
7    7          0.16690         1235
8    8          0.03130        34521
9    9          0.22204       146553
```

Read Data frames online: https://riptutorial.com/r/topic/438/data-frames

www.dbooks.org

Chapter 27: data.table

Introduction

Data.table is a package that extends the functionality of data frames from base R, particularly improving on their performance and syntax. See the package's Docs area at Getting started with data.table for details.

Syntax

- `DT[i, j, by]`
  # DT[where, select|update|do, by]
- `DT[...][...]`
  # chaining
- `################# Shortcuts, special functions and special symbols inside DT[...]`
- .()
  # in several arguments, replaces list()
- J()
  # in i, replaces list()
- :=
  # in j, a function used to add or modify columns
- .N
  # in i, the total number of rows
  # in j, the number of rows in a group
- .I
  # in j, the vector of row numbers in the table (filtered by i)
- .SD
  # in j, the current subset of the data
  # selected by the .SDcols argument
- .GRP
  # in j, the current index of the subset of the data
- .BY
  # in j, the list of by values for the current subset of data
- V1, V2, ...
  # default names for unnamed columns created in j
- `################# Joins inside DT[...]`
- DT1[DT2, on, j]
  # join two tables
- i.*
  # special prefix on DT2's columns after the join
- by=.EACHI
  # special option available only with a join
- DT1[!DT2, on, j]
  # anti-join two tables
- DT1[DT2, on, roll, j]

join two tables, rolling on the last column in on=

- `################ Reshaping, stacking and splitting`
- melt(DT, id.vars, measure.vars)
  # transform to long format
  # for multiple columns, use measure.vars = patterns(...)
- dcast(DT, formula)
  # transform to wide format
- rbind(DT1, DT2, ...)
  # stack enumerated data.tables
- rbindlist(DT_list, idcol)
  # stack a list of data.tables
- split(DT, by)
  # split a data.table into a list
- `################ Some other functions specialized for data.tables`
- foverlaps
  # overlap joins
- merge
  # another way of joining two tables
- set
  # another way of adding or modifying columns
- fintersect, fsetdiff, funion, fsetequal, unique, duplicated, anyDuplicated
  # set-theory operations with rows as elements
- uniqueN
  # the number of distinct rows
- rowidv(DT, cols)
  # row ID (1 to .N) within each group determined by cols
- rleidv(DT, cols)
  # group ID (1 to .GRP) within each group determined by runs of cols
- shift(DT, n, type=c("lag", "lead"))
  # apply a shift operator to every column
- setorder, setcolorder, setnames, setkey, setindex, setattr
  # modify attributes and order by reference

Remarks

Installation and support

To install the data.table package:

```
# install from CRAN
install.packages("data.table")

# or install development version
install.packages("data.table", type = "source", repos =
"http://Rdatatable.github.io/data.table")

# and to revert from devel to CRAN, the current version must first be removed
```


www.dbooks.org

```
remove.packages("data.table")
install.packages("data.table")
```

The package's official site has wiki pages providing help getting started, and lists of presentations and articles from around the web. Before asking a question -- here on StackOverflow or anywhere else -- please read the support page.

Loading the package

Many of the functions in the examples above exist in the data.table namespace. To use them, you will need to add a line like `library(data.table)` first or to use their full path, like `data.table::fread` instead of simply `fread`. For help on individual functions, the syntax is `help("fread")` or `?fread`. Again, if the package is not loaded, use the full name like `?data.table::fread`.

Examples

Creating a data.table

A data.table is an enhanced version of the data.frame class from base R. As such, its `class()` attribute is the vector `"data.table" "data.frame"` and functions that work on a data.frame will also work with a data.table. There are many ways to create, load or coerce to a data.table.

Build

Don't forget to install and activate the `data.table` package

```
library(data.table)
```

There is a constructor of the same name:

```
DT <- data.table(
  x = letters[1:5],
  y = 1:5,
  z = (1:5) > 3
)
#    x y     z
# 1: a 1 FALSE
# 2: b 2 FALSE
# 3: c 3 FALSE
# 4: d 4  TRUE
# 5: e 5  TRUE
```

Unlike `data.frame`, `data.table` will not coerce strings to factors:

```
sapply(DT, class)
#               x           y           z
#     "character"   "integer"   "logical"
```

Read in

We can read from a text file:

```
dt <- fread("my_file.csv")
```

Unlike `read.csv`, `fread` will read strings as strings, not as factors.

Modify a data.frame

For efficiency, data.table offers a way of altering a data.frame or list to make a data.table in-place (without making a copy or changing its memory location):

```
# example data.frame
DF <- data.frame(x = letters[1:5], y = 1:5, z = (1:5) > 3)
# modification
setDT(DF)
```

Note that we do not `<-` assign the result, since the object `DF` has been modified in-place. The class attributes of the data.frame will be retained:

```
sapply(DF, class)
#         x         y         z
#  "factor" "integer" "logical"
```

Coerce object to data.table

If you have a `list`, `data.frame`, or `data.table`, you should use the `setDT` function to convert to a `data.table` because it does the conversion by reference instead of making a copy (which `as.data.table` does). This is important if you are working with large datasets.

If you have another R object (such as a matrix), you must use `as.data.table` to coerce it to a `data.table`.

```
mat <- matrix(0, ncol = 10, nrow = 10)

DT <- as.data.table(mat)
# or
DT <- data.table(mat)
```

Adding and modifying columns

`DT[where, select|update|do, by]` syntax is used to work with columns of a data.table.

- The "where" part is the `i` argument

www.dbooks.org

- The "select|update|do" part is the `j` argument

These two arguments are usually passed by position instead of by name.

Our example data below is

```
mtcars = data.table(mtcars, keep.rownames = TRUE)
```

Editing entire columns

Use the `:=` operator inside `j` to assign new columns:

```
mtcars[, mpg_sq := mpg^2]
```

Remove columns by setting to `NULL`:

```
mtcars[, mpg_sq := NULL]
```

Add multiple columns by using the `:=` operator's multivariate format:

```
mtcars[, `:=`(mpg_sq = mpg^2, wt_sqrt = sqrt(wt))]
# or
mtcars[, c("mpg_sq", "wt_sqrt") := .(mpg^2, sqrt(wt))]
```

If the columns are dependent and must be defined in sequence, one way is:

```
mtcars[, c("mpg_sq", "mpg2_hp") := .(temp1 <- mpg^2, temp1/hp)]
```

The `.()` syntax is used when the right-hand side of `LHS := RHS` is a list of columns.

For dynamically-determined column names, use parentheses:

```
vn = "mpg_sq"
mtcars[, (vn) := mpg^2]
```

Columns can also be modified with `set`, though this is rarely necessary:

```
set(mtcars, j = "hp_over_wt", v = mtcars$hp/mtcars$wt)
```

Editing subsets of columns

Use the `i` argument to subset to rows "where" edits should be made:

```
mtcars[1:3, newvar := "Hello"]
# or
```

```
set(mtcars, j = "newvar", i = 1:3, v = "Hello")
```

As in a data.frame, we can subset using row numbers or logical tests. It is also possible to use a "join" in `i`, but that more complicated task is covered in another example.

Editing column attributes

Functions that edit attributes, such as `levels<-` or `names<-`, actually replace an object with a modified copy. Even if only used on one column in a data.table, the entire object is copied and replaced.

To modify an object without copies, use `setnames` to change the column names of a data.table or data.frame and `setattr` to change an attribute for any object.

```
# Print a message to the console whenever the data.table is copied
tracemem(mtcars)
mtcars[, cyl2 := factor(cyl)]

# Neither of these statements copy the data.table
setnames(mtcars, old = "cyl2", new = "cyl_fac")
setattr(mtcars$cyl_fac, "levels", c("four", "six", "eight"))

# Each of these statements copies the data.table
names(mtcars)[names(mtcars) == "cyl_fac"] <- "cf"
levels(mtcars$cf) <- c("IV", "VI", "VIII")
```

Be aware that these changes are made by reference, so they are *global*. Changing them within one environment affects the object in all environments.

```
# This function also changes the levels in the global environment
edit_levels <- function(x) setattr(x, "levels", c("low", "med", "high"))
edit_levels(mtcars$cyl_factor)
```

Special symbols in data.table

.SD

`.SD` refers to the subset of the `data.table` for each group, excluding all columns used in `by`.

`.SD` along with `lapply` can be used to apply any function to multiple columns by group in a `data.table`

We will continue using the same built-in dataset, `mtcars`:

```
mtcars = data.table(mtcars) # Let's not include rownames to keep things simpler
```

Mean of all columns in the dataset by *number of cylinders*, `cyl`:

www.dbooks.org

```
mtcars[ , lapply(.SD, mean), by = cyl]

#   cyl      mpg     disp        hp     drat       wt     qsec        vs        am     gear
carb
#1:   6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857143
3.428571
#2:   4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090909
1.545455
#3:   8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285714
3.500000
```

Apart from `cyl`, there are other categorical columns in the dataset such as `vs`, `am`, `gear` and `carb`. It doesn't really make sense to take the `mean` of these columns. So let's exclude these columns. This is where `.SDcols` comes into the picture.

.SDcols

`.SDcols` specifies the columns of the `data.table` that are included in `.SD`.

Mean of all columns (continuous columns) in the dataset by *number of gears* `gear`, and *number of cylinders*, `cyl`, arranged by `gear` and `cyl`:

```
# All the continuous variables in the dataset
cols_chosen <- c("mpg", "disp", "hp", "drat", "wt", "qsec")

mtcars[order(gear, cyl), lapply(.SD, mean), by = .(gear, cyl), .SDcols = cols_chosen]

#   gear cyl    mpg     disp       hp     drat       wt    qsec
#1:    3   4 21.500 120.1000  97.0000 3.700000 2.465000 20.0100
#2:    3   6 19.750 241.5000 107.5000 2.920000 3.337500 19.8300
#3:    3   8 15.050 357.6167 194.1667 3.120833 4.104083 17.1425
#4:    4   4 26.925 102.6250  76.0000 4.110000 2.378125 19.6125
#5:    4   6 19.750 163.8000 116.5000 3.910000 3.093750 17.6700
#6:    5   4 28.200 107.7000 102.0000 4.100000 1.826500 16.8000
#7:    5   6 19.700 145.0000 175.0000 3.620000 2.770000 15.5000
#8:    5   8 15.400 326.0000 299.5000 3.880000 3.370000 14.5500
```

Maybe we don't want to calculate the `mean` by groups. To calculate the mean for all the cars in the dataset, we don't specify the `by` variable.

```
mtcars[ , lapply(.SD, mean), .SDcols = cols_chosen]

#        mpg     disp       hp     drat      wt     qsec
#1: 20.09062 230.7219 146.6875 3.596563 3.21725 17.84875
```

Note:

- It is not necessary to define `cols_chosen` beforehand. `.SDcols` can directly take column names
- `.SDcols` can also directly take a vector of columnnumbers. In the above example this would be `mtcars[ , lapply(.SD, mean), .SDcols = c(1,3:7)]`

.N

`.N` is shorthand for the number of rows in a group.

```
iris[, .(count=.N), by=Species]

#      Species count
#1:     setosa    50
#2: versicolor    50
#3:  virginica    50
```

Writing code compatible with both data.frame and data.table

Differences in subsetting syntax

A `data.table` is one of several two-dimensional data structures available in R, besides `data.frame`, `matrix` and (2D) `array`. All of these classes use a very similar but not identical syntax for subsetting, the `A[rows, cols]` schema.

Consider the following data stored in a `matrix`, a `data.frame` and a `data.table`:

```
ma <- matrix(rnorm(12), nrow=4, dimnames=list(letters[1:4], c('X', 'Y', 'Z')))
df <- as.data.frame(ma)
dt <- as.data.table(ma)

ma[2:3]  #---> returns the 2nd and 3rd items, as if 'ma' were a vector (because it is!)
df[2:3]  #---> returns the 2nd and 3rd columns
dt[2:3]  #---> returns the 2nd and 3rd rows!
```

If you want to be sure of what will be returned, it is better to be *explicit*.

To get specific **rows**, just add a comma after the range:

```
ma[2:3, ]  # \
df[2:3, ]  #  }---> returns the 2nd and 3rd rows
dt[2:3, ]  # /
```

But, if you want to subset **columns**, some cases are interpreted differently. All three can be subset the same way with integer or character indices *not* stored in a variable.

```
ma[, 2:3]          #  \
df[, 2:3]          #   \
dt[, 2:3]          #    }---> returns the 2nd and 3rd columns
ma[, c("Y", "Z")]  #   /
df[, c("Y", "Z")]  #  /
dt[, c("Y", "Z")]  # /
```

However, they differ for unquoted variable names

www.dbooks.org

```
mycols <- 2:3
ma[, mycols]                 # \
df[, mycols]                 #  }---> returns the 2nd and 3rd columns
dt[, mycols, with = FALSE]   # /

dt[, mycols]                 # ---> Raises an error
```

In the last case, `mycols` is evaluated as the name of a column. Because `dt` cannot find a column named `mycols`, an error is raised.

Note: For versions of the `data.table` package priorto 1.9.8, this behavior was slightly different. Anything in the column index would have been evaluated using `dt` as an environment. So both `dt[, 2:3]` and `dt[, mycols]` would return the vector `2:3`. No error would be raised for the second case, because the variable `mycols` does exist in the parent environment.

Strategies for maintaining compatibility with data.frame and data.table

There are many reasons to write code that is guaranteed to work with `data.frame` and `data.table`. Maybe you are forced to use `data.frame`, or you may need to share some code that you don't know how will be used. So, there are some main strategies for achieving this, in order of convenience:

1. Use syntax that behaves the same for both classes.
2. Use a common function that does the same thing as the shortest syntax.
3. Force `data.table` to behave as `data.frame` (ex.: call the specific method `print.data.frame`).
4. Treat them as `list`, which they ultimately are.
5. Convert the table to a `data.frame` before doing anything (bad idea if it is a huge table).
6. Convert the table to `data.table`, if dependencies are not a concern.

Subset rows. Its simple, just use the `[, ]` selector, *with* the comma:

```
A[1:10, ]
A[A$var > 17, ]  # A[var > 17, ] just works for data.table
```

Subset columns. If you want a single column, use the `$` or the `[[ ]]` selector:

```
A$var
colname <- 'var'
A[[colname]]
A[[1]]
```

If you want a uniform way to grab more than one column, it's necessary to appeal a bit:

```
B <- `[.data.frame`(A, 2:4)

# We can give it a better name
select <- `[.data.frame`
B <- select(A, 2:4)
```

```
C <- select(A, c('foo', 'bar'))
```

Subset 'indexed' rows. While `data.frame` has `row.names`, `data.table` has its unique `key` feature. The best thing is to avoid `row.names` entirely and take advantage of the existing optimizations in the case of `data.table` when possible.

```
B <- A[A$var != 0, ]
# or...
B <- with(A, A[var != 0, ])  # data.table will silently index A by var before subsetting

stuff <- c('a', 'c', 'f')
C <- A[match(stuff, A$name), ]  # really worse than: setkey(A); A[stuff, ]
```

Get a 1-column table, get a row as a vector. These are easy with what we have seen until now:

```
B <- select(A, 2)    #---> a table with just the second column
C <- unlist(A[1, ])  #---> the first row as a vector (coerced if necessary)
```

Setting keys in data.table

Yes, you need to SETKEY pre 1.9.6

In the past (pre 1.9.6), your `data.table` was sped up by setting columns as keys to the table, particularly for large tables. [See intro vignette page 5 of September 2015 version, where speed of search was 544 times better.] You may find older code making use of this setting keys with 'setkey' or setting a 'key=' column when setting up the table.

```
library(data.table)
DT <- data.table(
  x = letters[1:5],
  y = 5:1,
  z = (1:5) > 3
)

#> DT
#   x y     z
#1: a 5 FALSE
#2: b 4 FALSE
#3: c 3 FALSE
#4: d 2  TRUE
#5: e 1  TRUE
```

Set your key with the `setkey` command. You can have a key with multiple columns.

```
setkey(DT, y)
```

Check your table's key in tables()

```
tables()

> tables()
     NAME NROW NCOL MB COLS  KEY
```


www.dbooks.org

```
[1,] DT      5    3  1 x,y,z y
Total: 1MB
```

Note this will re-sort your data.

```
#> DT
#   x y     z
#1: e 1  TRUE
#2: d 2  TRUE
#3: c 3 FALSE
#4: b 4 FALSE
#5: a 5 FALSE
```

Now it is unnecessary

Prior to v1.9.6 you had to have set a key for certain operations especially joining tables. The developers of data.table have sped up and introduced a `"on="` feature that can replace the dependency on keys. See SO answer here for a detailed discussion.

In Jan 2017, the developers have written a vignette around secondary indices which explains the "on" syntax and allows for other columns to be identified for fast indexing.

Creating secondary indices?

In a manner similar to key, you can `setindex(DT, key.col)` or `setindexv(DT, "key.col.string")`, where DT is your data.table. Remove all indices with `setindex(DT, NULL)`.

See your secondary indices with `indices(DT)`.

Why secondary indices?

This **does not sort** the table (unlike key), but does allow for quick indexing using the "on" syntax. Note there can be only one key, but you can use multiple secondary indices, which saves having to rekey and resort the table. This will speed up your subsetting when changing the columns you want to subset on.

Recall, in example above y was the key for table DT:

```
DT
# x y     z
# 1: e 1  TRUE
# 2: d 2  TRUE
# 3: c 3 FALSE
# 4: b 4 FALSE
# 5: a 5 FALSE

# Let us set x as index
setindex(DT, x)

# Use indices to see what has been set
indices(DT)
# [1] "x"

# fast subset using index and not keyed column
```

```
DT["c", on ="x"]
#x y     z
#1: c 3 FALSE

# old way would have been rekeying DT from y to x, doing subset and
# perhaps keying back to y (now we save two sorts)
# This is a toy example above but would have been more valuable with big data sets
```

Read data.table online: https://riptutorial.com/r/topic/849/data-table

www.dbooks.org

Chapter 28: Date and Time

Introduction

R comes with classes for dates, date-times and time differences; see `?Dates`, `?DateTimeClasses`, `?difftime` and follow the "See Also" section of those docs for further documentation. Related Docs: Dates and Date-Time Classes.

Remarks

Classes

- POSIXct

  A date-time class, POSIXct stores time as seconds since UNIX epoch on `1970-01-01 00:00:00 UTC`. It is the format returned when pulling the current time with `Sys.Time()`.

- POSIXlt

  A date-time class, stores a list of day, month, year, hour, minute, second, and so on. This is the format returned by `strptime`.

- Date The only date class, stores the date as a floating-point number.

Selecting a date-time format

POSIXct is the sole option in the tidyverse and world of UNIX. It is faster and takes up less memory than POSIXlt.

```
origin = as.POSIXct("1970-01-01 00:00:00", format ="%Y-%m-%d %H:%M:%S", tz = "UTC")

origin
## [1] "1970-01-01 UTC"

origin + 47
## [1] "1970-01-01 00:00:47 UTC"

as.numeric(origin)      # At epoch
## 0

as.numeric(Sys.time()) # Right now (output as of July 21, 2016 at 11:47:37 EDT)
## 1469116057

posixlt = as.POSIXlt(Sys.time(), format ="%Y-%m-%d %H:%M:%S", tz = "America/Chicago")

# Conversion to POISXct
posixct = as.POSIXct(posixlt)
```

```
posixct

# Accessing components
posixlt$sec   # Seconds 0-61
posixlt$min   # Minutes 0-59
posixlt$hour  # Hour 0-23
posixlt$mday  # Day of the Month 1-31
posixlt$mon   # Months after the first of the year 0-11
posixlt$year  # Years since 1900.

ct = as.POSIXct("2015-05-25")
lt = as.POSIXlt("2015-05-25")

object.size(ct)
# 520 bytes
object.size(lt)
# 1816 bytes
```

Specialized packages

- anytime
- data.table IDate and ITime
- fasttime
- lubridate
- nanotime

Examples

Current Date and Time

R is able to access the current date, time and time zone:

```
Sys.Date()             # Returns date as a Date object

## [1] "2016-07-21"

Sys.time()             # Returns date & time at current locale as a POSIXct object

## [1] "2016-07-21 10:04:39 CDT"

as.numeric(Sys.time()) # Seconds from UNIX Epoch (1970-01-01 00:00:00 UTC)

## [1] 1469113479

Sys.timezone()         # Time zone at current location

## [1] "Australia/Melbourne"
```

Use `OlsonNames()` to view the time zone names in Olson/IANA database on the current system:

```
str(OlsonNames())
## chr [1:589] "Africa/Abidjan" "Africa/Accra" "Africa/Addis_Ababa" "Africa/Algiers"
```


www.dbooks.org

```
"Africa/Asmara" "Africa/Asmera" "Africa/Bamako" ...
```

Go to the End of the Month

Let's say we want to go to the last day of the month, this function will help on it:

```
eom <- function(x, p=as.POSIXlt(x)) as.Date(modifyList(p, list(mon=p$mon + 1, mday=0)))
```

Test:

```
x <- seq(as.POSIXct("2000-12-10"),as.POSIXct("2001-05-10"),by="months")
> data.frame(before=x,after=eom(x))
      before      after
1 2000-12-10 2000-12-31
2 2001-01-10 2001-01-31
3 2001-02-10 2001-02-28
4 2001-03-10 2001-03-31
5 2001-04-10 2001-04-30
6 2001-05-10 2001-05-31
>
```

Using a date in a string format:

```
> eom('2000-01-01')
[1] "2000-01-31"
```

Go to First Day of the Month

Let's say we want to go to the first day of a given month:

```
date <- as.Date("2017-01-20")

> as.POSIXlt(cut(date, "month"))
[1] "2017-01-01 EST"
```

Move a date a number of months consistently by months

Let's say we want to move a given date a `num`of months. We can define the following function, that uses the `mondate` package:

```
moveNumOfMonths <- function(date, num) {
    as.Date(mondate(date) + num)
}
```

It moves consistently the month part of the date and adjusting the day, in case the date refers to the last day of the month.

For example:

Back one month:

```
> moveNumOfMonths("2017-10-30",-1)
[1] "2017-09-30"
```

Back two months:

```
> moveNumOfMonths("2017-10-30",-2)
[1] "2017-08-30"
```

Forward two months:

```
> moveNumOfMonths("2017-02-28", 2)
[1] "2017-04-30"
```

It moves two months from the last day of February, therefore the last day of April.

Let's se how it works for backward and forward operations when it is the last day of the month:

```
> moveNumOfMonths("2016-11-30", 2)
[1] "2017-01-31"
> moveNumOfMonths("2017-01-31", -2)
[1] "2016-11-30"
```

Because November has 30 days, we get the same date in the backward operation, but:

```
> moveNumOfMonths("2017-01-30", -2)
[1] "2016-11-30"
> moveNumOfMonths("2016-11-30", 2)
[1] "2017-01-31"
```

Because January has 31 days, then moving two months from last day of November will get the last day of January.

Read Date and Time online: https://riptutorial.com/r/topic/1157/date-and-time

www.dbooks.org

Chapter 29: Date-time classes (POSIXct and POSIXlt)

Introduction

R includes two date-time classes -- POSIXct and POSIXlt -- see `?DateTimeClasses`.

Remarks

Pitfalls

With POSIXct, midnight will display only the date and time zone, though the full time is still stored.

Related topics

- Date and Time

Specialized packages

- lubridate

Examples

Formatting and printing date-time objects

```
# test date-time object
options(digits.secs = 3)
d = as.POSIXct("2016-08-30 14:18:30.58", tz = "UTC")

format(d,"%S")  # 00-61 Second as integer
## [1] "30"

format(d,"%OS") # 00-60.99… Second as fractional
## [1] "30.579"

format(d,"%M")  # 00-59 Minute
## [1] "18"

format(d,"%H")  # 00-23 Hours
## [1] "14"

format(d,"%I")  # 01-12 Hours
## [1] "02"
```

```
format(d,"%p")  # AM/PM Indicator
## [1] "PM"

format(d,"%z")  # Signed offset
## [1] "+0000"

format(d,"%Z")  # Time Zone Abbreviation
## [1] "UTC"
```

See `?strptime` for details on the format strings here, as well as other formats.

Parsing strings into date-time objects

The functions for parsing a string into POSIXct and POSIXlt take similar parameters and return a similar-looking result, but there are differences in how that date-time is stored; see "Remarks."

```
as.POSIXct("11:38",                         # time string
           format = "%H:%M")                # formatting string
## [1] "2016-07-21 11:38:00 CDT"
strptime("11:38",                           # identical, but makes a POSIXlt object
         format = "%H:%M")
## [1] "2016-07-21 11:38:00 CDT"

as.POSIXct("11 AM",
           format = "%I %p")
## [1] "2016-07-21 11:00:00 CDT"
```

Note that date and timezone are imputed.

```
as.POSIXct("11:38:22",                  # time string without timezone
           format = "%H:%M:%S",
           tz = "America/New_York")     # set time zone
## [1] "2016-07-21 11:38:22 EDT"

as.POSIXct("2016-07-21 00:00:00",
           format = "%F %T")            # shortcut tokens for "%Y-%m-%d" and "%H:%M:%S"
```

See `?strptime` for details on the format strings here.

Notes

Missing elements

- If a date element is not supplied, then that from the current date is used.
- If a time element is not supplied, then that from midnight is used, i.e. 0s.
- If no timezone is supplied in either the string or the `tz` parameter, the local timezone is used.

Time zones

www.dbooks.org

- The accepted values of `tz` depend on the location.
  - `CST` is given with `"CST6CDT"` or `"America/Chicago"`
- For supported locations and time zones use:
  - In R: `OlsonNames()`
  - Alternatively, try in R: `system("cat $R_HOME/share/zoneinfo/zone.tab")`
- These locations are given by Internet Assigned Numbers Authority (IANA)
  - List of tz database time zones (Wikipedia)
  - IANA TZ Data (2016e)

Date-time arithmetic

To add/subtract time, use POSIXct, since it stores times in seconds

```
## adding/subtracting times - 60 seconds
as.POSIXct("2016-01-01") + 60
# [1] "2016-01-01 00:01:00 AEDT"

## adding 3 hours, 14 minutes, 15 seconds
as.POSIXct("2016-01-01") + ( (3 * 60 * 60) + (14 * 60) + 15)
# [1] "2016-01-01 03:14:15 AEDT"
```

More formally, `as.difftime` can be used to specify time periods to add to a date or datetime object. E.g.:

```
as.POSIXct("2016-01-01")         +
  as.difftime(3,  units="hours") +
  as.difftime(14, units="mins")  +
  as.difftime(15, units="secs")
# [1] "2016-01-01 03:14:15 AEDT"
```

To find the difference between dates/times use `difftime()` for differences in seconds, minutes, hours, days or weeks.

```
# using POSIXct objects
difftime(
  as.POSIXct("2016-01-01 12:00:00"),
  as.POSIXct("2016-01-01 11:59:59"),
  unit = "secs")
# Time difference of 1 secs
```

To generate sequences of date-times use `seq.POSIXt()` or simply `seq`.

Read Date-time classes (POSIXct and POSIXlt) online: https://riptutorial.com/r/topic/9027/date-time-classes--posixct-and-posixlt-

Chapter 30: Debugging

Examples

Using browser

The `browser` function can be used like a breakpoint: code execution will pause at the point it is called. Then user can then inspect variable values, execute arbitrary R code and step through the code line by line.

Once `browser()` is hit in the code the interactive interpreter will start. Any R code can be run as normal, and in addition the following commands are present,

Command	Meaning
c	Exit browser and continue program
f	Finish current loop or function \
n	Step Over (evaluate next statement, stepping over function calls)
s	Step Into (evaluate next statement, stepping into function calls)
where	Print stack trace
r	Invoke "resume" restart
Q	Exit browser and quit

For example we might have a script like,

```
toDebug <- function() {
    a = 1
    b = 2

    browser()

    for(i in 1:100) {
        a = a * b
    }
}

toDebug()
```

When running the above script we initially see something like,

```
Called from: toDebug
Browser[1]>
```


www.dbooks.org

We could then interact with the prompt as so,

```
Called from: toDebug
Browser[1]> a
[1] 1
Browser[1]> b
[1] 2
Browse[1]> n
debug at #7: for (i in 1:100) {
    a = a * b
}
Browse[2]> n
debug at #8: a = a * b
Browse[2]> a
[1] 1
Browse[2]> n
debug at #8: a = a * b
Browse[2]> a
[1] 2
Browse[2]> Q
```

`browser()` can also be used as part of a functional chain, like so:

```
mtcars %>% group_by(cyl) %>% {browser()}
```

Using debug

You can set any function for debugging with `debug`.

```
debug(mean)
mean(1:3)
```

All subsequent calls to the function will enter debugging mode. You can disable this behavior with `undebug`.

```
undebug(mean)
mean(1:3)
```

If you know you only want to enter the debugging mode of a function once, consider the use of `debugonce`.

```
debugonce(mean)
mean(1:3)
mean(1:3)
```

Read Debugging online: https://riptutorial.com/r/topic/1695/debugging

Chapter 31: Distribution Functions

Introduction

R has many built-in functions to work with probability distributions, with official docs starting at `?Distributions`.

Remarks

There are generally four prefixes:

- **d**-The **density** function for the given distribution
- **p**-The cumulative distribution function
- **q**-Get the **quantile** associated with the given probability
- **r**-Get a **random** sample

For the distributions built into R's base installation, see `?Distributions`.

Examples

Normal distribution

Let's use `*norm` as an example. From the documentation:

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

So if I wanted to know the value of a standard normal distribution at 0, I would do

```
dnorm(0)
```

Which gives us `0.3989423`, a reasonable answer.

In the same way `pnorm(0)` gives `.5`. Again, this makes sense, because half of the distribution is to the left of 0.

`qnorm` will essentially do the opposite of `pnorm`. `qnorm(.5)` gives `0`.

Finally, there's the `rnorm` function:

```
rnorm(10)
```

Will generate 10 samples from standard normal.

www.dbooks.org

If you want to change the parameters of a given distribution, simply change them like so

```
rnorm(10, mean=4, sd= 3)
```

Binomial Distribution

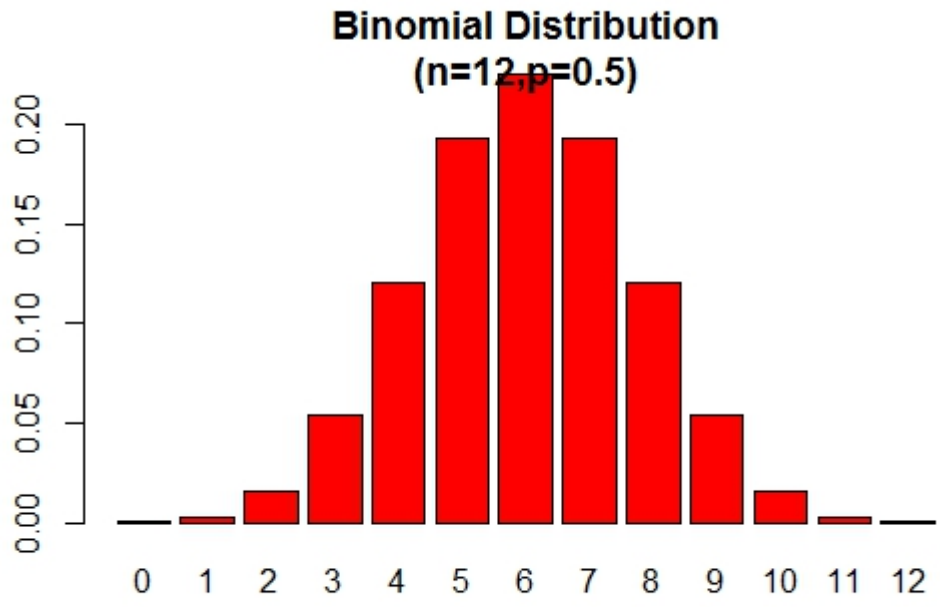
We now illustrate the functions `dbinom`,`pbinom`,`qbinom` and `rbinom` defined for *Binomial distribution*.

The `dbinom()` function gives the probabilities for various values of the binomial variable. Minimally it requires three arguments. The first argument for this function must be a vector of quantiles(the possible values of the random variable `x`). The second and third arguments are the `defining parameters` of the distribution, namely, `n`(the number of independent trials) and `p`(the probability of success in each trial). For example, for a binomial distribution with `n = 5`, `p = 0.5`, the possible values for X are `0,1,2,3,4,5`. That is, the `dbinom(x,n,p)` function gives the probability values `P( X = x )` for `x = 0, 1, 2, 3, 4, 5`.

```
#Binom(n = 5, p = 0.5) probabilities
> n <- 5; p<- 0.5; x <- 0:n
> dbinom(x,n,p)
[1] 0.03125 0.15625 0.31250 0.31250 0.15625 0.03125
#To verify the total probability is 1
> sum(dbinom(x,n,p))
[1] 1
>
```

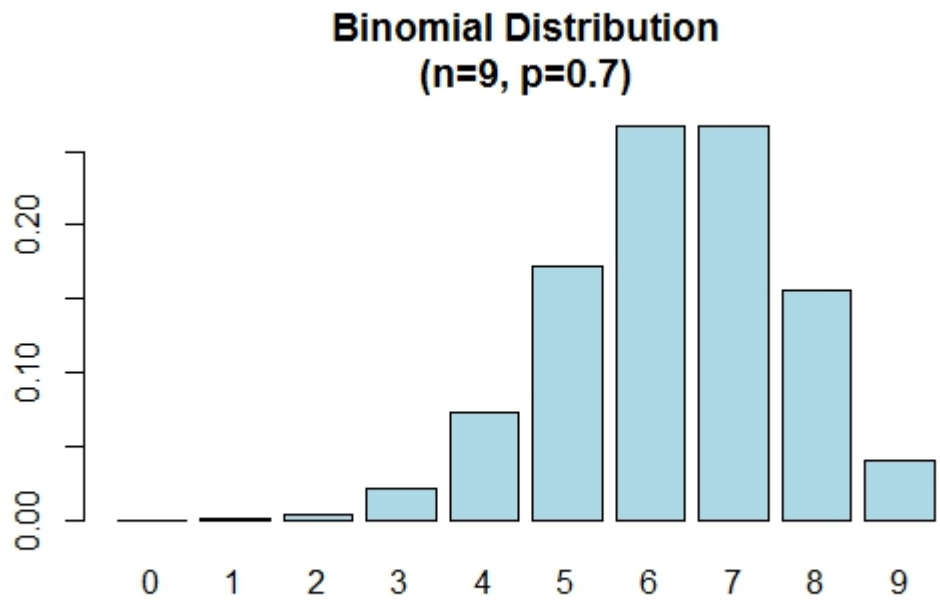
The binomial probability distribution plot can be displayed as in the following figure:

```
> x <- 0:12
> prob <- dbinom(x,12,.5)
> barplot(prob,col = "red",ylim = c(0,.2),names.arg=x,
                           main="Binomial Distribution\n(n=12,p=0.5)")
```



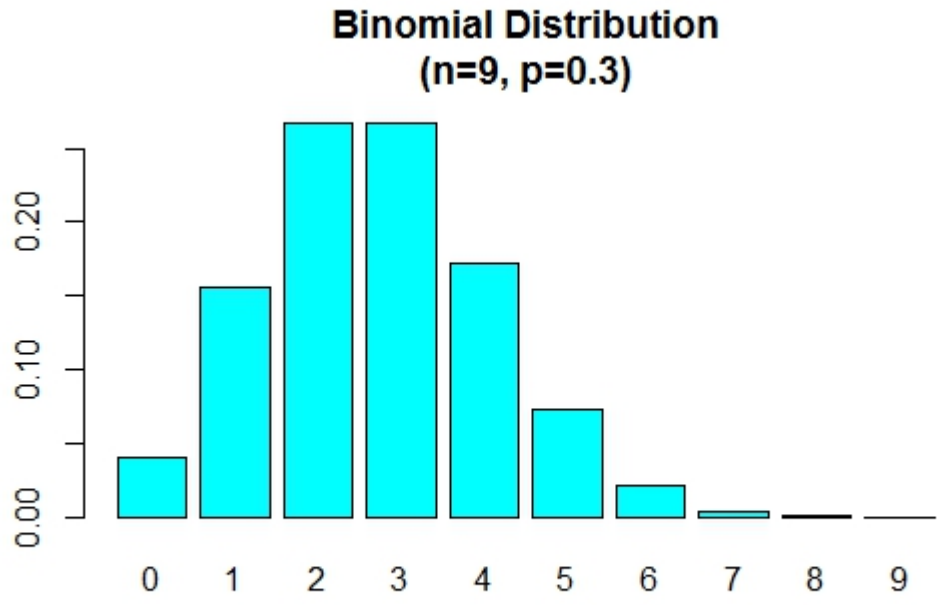
Note that the binomial distribution is symmetric when `p = 0.5`. To demonstrate that the binomial distribution is negatively skewed when `p` is larger than `0.5`, consider the following example:

```
> n=9; p=.7; x=0:n; prob=dbinom(x,n,p);
> barplot(prob,names.arg = x,main="Binomial Distribution\n(n=9, p=0.7)",col="lightblue")
```



When `p` is smaller than `0.5` the binomial distribution is positively skewed as shown below.

```
> n=9; p=.3; x=0:n; prob=dbinom(x,n,p);
> barplot(prob,names.arg = x,main="Binomial Distribution\n(n=9, p=0.3)",col="cyan")
```



www.dbooks.org

We will now illustrate the usage of the cumulative distribution function `pbinom()`. This function can be used to calculate probabilities such as `P( X <= x )`. The first argument to this function is a vector of quantiles(values of x).

```
# Calculating Probabilities
# P(X <= 2) in a Bin(n=5,p=0.5) distribution
> pbinom(2,5,0.5)
[1] 0.5
```

The above probability can also be obtained as follows:

```
# P(X <= 2) = P(X=0) + P(X=1) + P(X=2)
> sum(dbinom(0:2,5,0.5))
[1] 0.5
```

To compute, probabilities of the type: `P( a <= X <= b )`

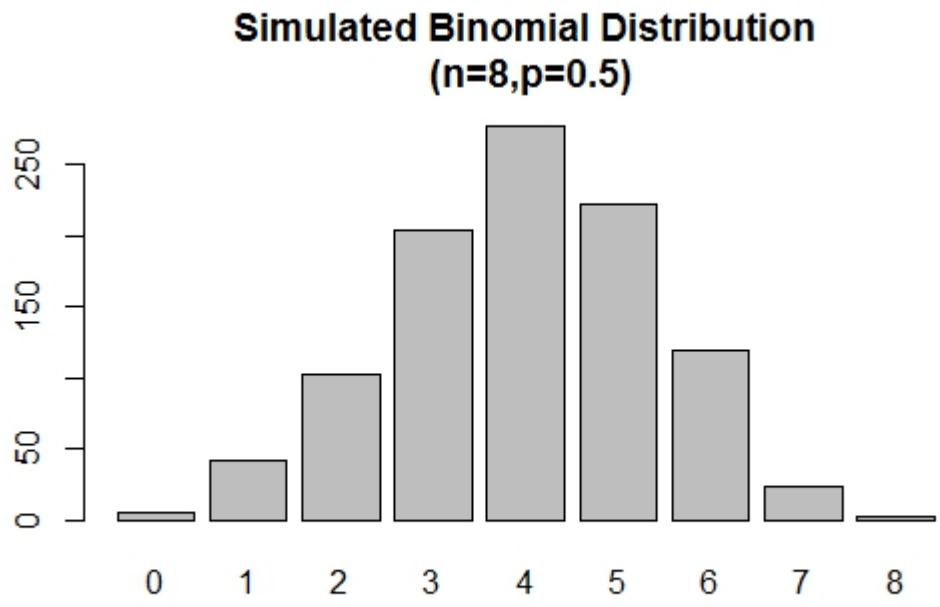
```
# P(3<= X <= 5) = P(X=3) + P(X=4) + P(X=5) in a Bin(n=9,p=0.6) dist
> sum(dbinom(c(3,4,5),9,0.6))
[1] 0.4923556
>
```

Presenting the binomial distribution in the form of a table:

```
> n = 10; p = 0.4; x = 0:n;
> prob = dbinom(x,n,p)
> cdf = pbinom(x,n,p)
> distTable = cbind(x,prob,cdf)
> distTable
       x         prob         cdf
 [1,]  0 0.0060466176 0.006046618
 [2,]  1 0.0403107840 0.046357402
 [3,]  2 0.1209323520 0.167289754
 [4,]  3 0.2149908480 0.382280602
 [5,]  4 0.2508226560 0.633103258
 [6,]  5 0.2006581248 0.833761382
 [7,]  6 0.1114767360 0.945238118
 [8,]  7 0.0424673280 0.987705446
 [9,]  8 0.0106168320 0.998322278
[10,]  9 0.0015728640 0.999895142
[11,] 10 0.0001048576 1.000000000
>
```

The `rbinom()` is used to generate random samples of specified sizes with a given parameter values.

```
# Simulation
> xVal<-names(table(rbinom(1000,8,.5)))
> barplot(as.vector(table(rbinom(1000,8,.5))),names.arg =xVal,
                    main="Simulated Binomial Distribution\n (n=8,p=0.5)")
```



Read Distribution Functions online: https://riptutorial.com/r/topic/1885/distribution-functions

www.dbooks.org

Chapter 32: dplyr

Remarks

dplyr is an iteration of plyr that provides a flexible "verb" based functions to manipulate data in R. The latest version of dplyr can be downloaded from CRAN using

```
install.package("dplyr")
```

The key object in dplyr is a tbl, a representation of a tabular data structure. Currently dplyr (version 0.5.0) supports:

- data frames
- data tables
- SQLite
- PostgreSQL/Redshift
- MySQL/MariaDB
- Bigquery
- MonetDB
- data cubes with arrays (partial implementation)

Examples

dplyr's single table verbs

dplyr introduces a grammar of data manipulation in `R`. It provides a consistent interface to work with data no matter where it is stored: data.frame, data.table, or a `database`. The key pieces of `dplyr` are written using Rcpp, which makes it very fast for working with in-memory data.

`dplyr`'s philosophy is to have small functions that do one thing well. The five simple functions (`filter`, `arrange`, `select`, `mutate`, and `summarise`) can be used to reveal new ways to describe data. When combined with `group_by`, these functions can be used to calculate group wise summary statistics.

Syntax commonalities

All these functions have a similar syntax:

- The first argument to all these functions is always a data frame
- Columns can be referred directly using bare variable names (i.e., without using `$`)
- These functions do not modify the original data itself, i.e., they don't have side effects. Hence, the results should always be saved to an object.

We will use the built-in mtcars dataset to explore `dplyr`'s single table verbs. Before converting the type of `mtcars` to `tbl_df` (since it makes printing cleaner), we add the `rownames` of the dataset as a

column using `rownames_to_column` function from the tibble package.

```
library(dplyr) # This documentation was written using version 0.5.0

mtcars_tbl <- as_data_frame(tibble::rownames_to_column(mtcars, "cars"))

# examine the structure of data
head(mtcars_tbl)

# A tibble: 6 x 12
#               cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#              <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1         Mazda RX4  21.0     6   160   110  3.90 2.620 16.46     0     1     4     4
#2     Mazda RX4 Wag  21.0     6   160   110  3.90 2.875 17.02     0     1     4     4
#3        Datsun 710  22.8     4   108    93  3.85 2.320 18.61     1     1     4     1
#4    Hornet 4 Drive  21.4     6   258   110  3.08 3.215 19.44     1     0     3     1
#5 Hornet Sportabout  18.7     8   360   175  3.15 3.440 17.02     0     0     3     2
#6           Valiant  18.1     6   225   105  2.76 3.460 20.22     1     0     3     1
```

filter

`filter` helps subset rows that match certain criteria. The first argument is the name of the `data.frame` and the second (and subsequent) arguments are the criteria that filter the data (these criteria should evaluate to either `TRUE` or `FALSE`)

Subset all cars that have 4 *cylinders* - `cyl`:

```
filter(mtcars_tbl, cyl == 4)

# A tibble: 11 x 12
#             cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#            <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1      Datsun 710  22.8     4 108.0    93  3.85 2.320 18.61     1     1     4     1
#2       Merc 240D  24.4     4 146.7    62  3.69 3.190 20.00     1     0     4     2
#3        Merc 230  22.8     4 140.8    95  3.92 3.150 22.90     1     0     4     2
#4        Fiat 128  32.4     4  78.7    66  4.08 2.200 19.47     1     1     4     1
#5     Honda Civic  30.4     4  75.7    52  4.93 1.615 18.52     1     1     4     2
# ... with 6 more rows
```

We can pass multiple criteria separated by a comma. To subset the cars which have either 4 or 6 *cylinders* - `cyl` and have 5 *gears* - `gear`:

```
filter(mtcars_tbl, cyl == 4 | cyl == 6, gear == 5)

# A tibble: 3 x 12
#           cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#          <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1 Porsche 914-2  26.0     4 120.3    91  4.43 2.140  16.7     0     1     5     2
#2  Lotus Europa  30.4     4  95.1   113  3.77 1.513  16.9     1     1     5     2
#3  Ferrari Dino  19.7     6 145.0   175  3.62 2.770  15.5     0     1     5     6
```

`filter` selects rows based on criteria, to select rows by position, use `slice`. `slice` takes only 2 arguments: the first one is a `data.frame` and the second is integer row values.

www.dbooks.org

To select rows 6 through 9:

```
slice(mtcars_tbl, 6:9)

# A tibble: 4 x 12
#        cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#       <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1    Valiant  18.1     6 225.0   105  2.76  3.46 20.22     1     0     3     1
#2 Duster 360  14.3     8 360.0   245  3.21  3.57 15.84     0     0     3     4
#3  Merc 240D  24.4     4 146.7    62  3.69  3.19 20.00     1     0     4     2
#4   Merc 230  22.8     4 140.8    95  3.92  3.15 22.90     1     0     4     2
```

Or:

```
slice(mtcars_tbl, -c(1:5, 10:n()))
```

This results in the same output as `slice(mtcars_tbl, 6:9)`

`n()` represents the number of observations in the current group

arrange

`arrange` is used to sort the data by a specified variable(s). Just like the previous verb (and all other functions in `dplyr`), the first argument is a `data.frame`, and consequent arguments are used to sort the data. If more than one variable is passed, the data is first sorted by the first variable, and then by the second variable, and so on..

To order the data by *horsepower* - `hp`

```
arrange(mtcars_tbl, hp)

# A tibble: 32 x 12
#             cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#            <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1     Honda Civic  30.4     4  75.7    52  4.93 1.615 18.52     1     1     4     2
#2       Merc 240D  24.4     4 146.7    62  3.69 3.190 20.00     1     0     4     2
#3  Toyota Corolla  33.9     4  71.1    65  4.22 1.835 19.90     1     1     4     1
#4        Fiat 128  32.4     4  78.7    66  4.08 2.200 19.47     1     1     4     1
#5       Fiat X1-9  27.3     4  79.0    66  4.08 1.935 18.90     1     1     4     1
#6   Porsche 914-2  26.0     4 120.3    91  4.43 2.140 16.70     0     1     5     2
# ... with 26 more rows
```

To `arrange` the data by *miles per gallon* - `mpg` in descending order, followed by *number of cylinders* - `cyl`:

```
arrange(mtcars_tbl, desc(mpg), cyl)

# A tibble: 32 x 12
#             cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#            <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1  Toyota Corolla  33.9     4  71.1    65  4.22 1.835 19.90     1     1     4     1
#2        Fiat 128  32.4     4  78.7    66  4.08 2.200 19.47     1     1     4     1
```

```
#3     Honda Civic  30.4     4  75.7    52  4.93 1.615 18.52     1     1     4     2
#4    Lotus Europa  30.4     4  95.1   113  3.77 1.513 16.90     1     1     5     2
#5       Fiat X1-9  27.3     4  79.0    66  4.08 1.935 18.90     1     1     4     1
#6   Porsche 914-2  26.0     4 120.3    91  4.43 2.140 16.70     0     1     5     2
# ... with 26 more rows
```

select

`select` is used to select only a subset of variables. To select only `mpg`, `disp`, `wt`, `qsec`, and `vs` from `mtcars_tbl`:

```
select(mtcars_tbl, mpg, disp, wt, qsec, vs)

# A tibble: 32 x 5
#     mpg  disp    wt  qsec    vs
#   <dbl> <dbl> <dbl> <dbl> <dbl>
#1   21.0 160.0 2.620 16.46     0
#2   21.0 160.0 2.875 17.02     0
#3   22.8 108.0 2.320 18.61     1
#4   21.4 258.0 3.215 19.44     1
#5   18.7 360.0 3.440 17.02     0
#6   18.1 225.0 3.460 20.22     1
# ... with 26 more rows
```

`:` notation can be used to select consecutive columns. To select columns from `cars` through `disp` and `vs` through `carb`:

```
select(mtcars_tbl, cars:disp, vs:carb)

# A tibble: 32 x 8
#                cars   mpg   cyl  disp    vs    am  gear  carb
#               <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1          Mazda RX4  21.0     6 160.0     0     1     4     4
#2      Mazda RX4 Wag  21.0     6 160.0     0     1     4     4
#3         Datsun 710  22.8     4 108.0     1     1     4     1
#4     Hornet 4 Drive  21.4     6 258.0     1     0     3     1
#5  Hornet Sportabout  18.7     8 360.0     0     0     3     2
#6            Valiant  18.1     6 225.0     1     0     3     1
# ... with 26 more rows
```

or `select(mtcars_tbl, -(hp:qsec))`

For datasets that contain several columns, it can be tedious to select several columns by name. To make life easier, there are a number of helper functions (such as `starts_with()`, `ends_with()`, `contains()`, `matches()`, `num_range()`, `one_of()`, and `everything()`) that can be used in `select`. To learn more about how to use them, see `?select_helpers` and `?select`.

Note: While referring to columns directly in `select()`, we use bare column names, but quotes should be used while referring to columns in helper functions.

To rename columns while selecting:

www.dbooks.org

```
select(mtcars_tbl, cylinders = cyl, displacement = disp)

# A tibble: 32 x 2
#   cylinders displacement
#       <dbl>        <dbl>
#1          6        160.0
#2          6        160.0
#3          4        108.0
#4          6        258.0
#5          8        360.0
#6          6        225.0
# ... with 26 more rows
```

As expected, this drops all other variables.

To rename columns without dropping other variables, use `rename`:

```
rename(mtcars_tbl, cylinders = cyl, displacement = disp)

# A tibble: 32 x 12
#                cars   mpg cylinders displacement    hp  drat    wt  qsec    vs
#               <chr> <dbl>     <dbl>        <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#1          Mazda RX4  21.0         6        160.0   110  3.90 2.620 16.46     0
#2      Mazda RX4 Wag  21.0         6        160.0   110  3.90 2.875 17.02     0
#3         Datsun 710  22.8         4        108.0    93  3.85 2.320 18.61     1
#4     Hornet 4 Drive  21.4         6        258.0   110  3.08 3.215 19.44     1
#5  Hornet Sportabout  18.7         8        360.0   175  3.15 3.440 17.02     0
#6            Valiant  18.1         6        225.0   105  2.76 3.460 20.22     1
# ... with 26 more rows, and 3 more variables: am <dbl>, gear <dbl>, carb <dbl>
```

mutate

`mutate` can be used to add new columns to the data. Like all other functions in `dplyr`, mutate doesn't add the newly created columns to the original data. Columns are added at the end of the `data.frame`.

```
mutate(mtcars_tbl, weight_ton = wt/2, weight_pounds = weight_ton * 2000)

# A tibble: 32 x 14
#                cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
weight_ton weight_pounds
#               <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
<dbl>         <dbl>
#1          Mazda RX4  21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
1.3100          2620
#2      Mazda RX4 Wag  21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
1.4375          2875
#3         Datsun 710  22.8     4 108.0    93  3.85 2.320 18.61     1     1     4     1
1.1600          2320
#4     Hornet 4 Drive  21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
1.6075          3215
#5  Hornet Sportabout  18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
1.7200          3440
#6            Valiant  18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
1.7300          3460
```

```
# ... with 26 more rows
```

Note the use of `weight_ton` while creating `weight_pounds`. Unlike base `R`, `mutate` allows us to refer to columns that we just created to be used for a subsequent operation.

To retain only the newly created columns, use `transmute` instead of `mutate`:

```
transmute(mtcars_tbl, weight_ton = wt/2, weight_pounds = weight_ton * 2000)

# A tibble: 32 x 2
#   weight_ton weight_pounds
#        <dbl>         <dbl>
#1      1.3100          2620
#2      1.4375          2875
#3      1.1600          2320
#4      1.6075          3215
#5      1.7200          3440
#6      1.7300          3460
# ... with 26 more rows
```

summarise

`summarise` calculates summary statistics of variables by collapsing multiple values to a single value. It can calculate multiple statistics and we can name these summary columns in the same statement.

To calculate the *mean* and *standard deviation* of `mpg` and `disp` of all cars in the dataset:

```
summarise(mtcars_tbl, mean_mpg = mean(mpg), sd_mpg = sd(mpg),
          mean_disp = mean(disp), sd_disp = sd(disp))

# A tibble: 1 x 4
#  mean_mpg   sd_mpg mean_disp  sd_disp
#     <dbl>    <dbl>     <dbl>    <dbl>
#1 20.09062 6.026948  230.7219 123.9387
```

group_by

`group_by` can be used to perform group wise operations on data. When the verbs defined above are applied on this grouped data, they are automatically applied to each group separately.

To find *`mean`* and *`sd`* of *`mpg`* by *`cyl`*:

```
by_cyl <- group_by(mtcars_tbl, cyl)
summarise(by_cyl, mean_mpg = mean(mpg), sd_mpg = sd(mpg))


# A tibble: 3 x 3
#    cyl mean_mpg   sd_mpg
#  <dbl>    <dbl>    <dbl>
```


www.dbooks.org

```
#1     4 26.66364 4.509828
#2     6 19.74286 1.453567
#3     8 15.10000 2.560048
```

Putting it all togther

We select columns from `cars` through `hp` and `gear`, order the rows by `cyl` and from highest to lowest `mpg`, group the data by `gear`, and finally subset only those cars have `mpg` > 20 and `hp` > 75

```
selected <- select(mtcars_tbl, cars:hp, gear)
ordered <- arrange(selected, cyl, desc(mpg))
by_cyl <- group_by(ordered, gear)
filter(by_cyl, mpg > 20, hp > 75)

Source: local data frame [9 x 6]
Groups: gear [3]

#            cars   mpg   cyl  disp    hp  gear
#           <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
#1   Lotus Europa  30.4     4  95.1   113     5
#2  Porsche 914-2  26.0     4 120.3    91     5
#3     Datsun 710  22.8     4 108.0    93     4
#4       Merc 230  22.8     4 140.8    95     4
#5  Toyota Corona  21.5     4 120.1    97     3
# ... with 4 more rows
```

Maybe we are not interested the intermediate results, we can achieve the same result as above by wrapping the function calls:

```
filter(
    group_by(
        arrange(
            select(
                mtcars_tbl, cars:hp
            ), cyl, desc(mpg)
        ), cyl
    ),mpg > 20, hp > 75
)
```

This can be a little difficult to read. So, `dplyr` operations can be chained using the pipe `%>%` operator. The above code transalates to:

```
mtcars_tbl %>%
    select(cars:hp) %>%
    arrange(cyl, desc(mpg)) %>%
    group_by(cyl) %>%
    filter(mpg > 20, hp > 75)
```

summarise multiple columns

`dplyr` provides `summarise_all()` to apply functions to all (non-grouping) columns.

To find the number of distinct values for each column:

```
mtcars_tbl %>%
    summarise_all(n_distinct)

# A tibble: 1 x 12
#   cars   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
#  <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
#1    32    25     3    27    22    22    29    30     2     2     3     6
```

To find the number of distinct values for each column by `cyl`:

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_all(n_distinct)

# A tibble: 3 x 12
#    cyl  cars   mpg  disp    hp  drat    wt  qsec    vs    am  gear  carb
#  <dbl> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int> <int>
#1     4    11     9    11    10    10    11    11     2     2     3     2
#2     6     7     6     5     4     5     6     7     2     2     3     3
#3     8    14    12    11     9    11    13    14     1     2     2     4
```

Note that we just had to add the `group_by` statement and the rest of the code is the same. The output now consists of three rows - one for each unique value of `cyl`.

To `summarise` specific multiple columns, use `summarise_at`

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_at(c("mpg", "disp", "hp"), mean)

# A tibble: 3 x 4
#    cyl      mpg     disp        hp
#  <dbl>    <dbl>    <dbl>     <dbl>
#1     4 26.66364 105.1364  82.63636
#2     6 19.74286 183.3143 122.28571
#3     8 15.10000 353.1000 209.21429
```

`helper` functions (`?select_helpers`) can be used in place of column names to select specific columns

To apply multiple functions, either pass the function names as a character vector:

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_at(c("mpg", "disp", "hp"),
                 c("mean", "sd"))
```

or wrap them inside `funs`:

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_at(c("mpg", "disp", "hp"),
```


www.dbooks.org

```
                funs(mean, sd))

# A tibble: 3 x 7
#    cyl mpg_mean disp_mean   hp_mean   mpg_sd  disp_sd    hp_sd
#  <dbl>    <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
#1     4 26.66364  105.1364  82.63636 4.509828 26.87159 20.93453
#2     6 19.74286  183.3143 122.28571 1.453567 41.56246 24.26049
#3     8 15.10000  353.1000 209.21429 2.560048 67.77132 50.97689
```

Column names are now be appended with function names to keep them distinct. In order to change this, pass the name to be appended with the function:

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_at(c("mpg", "disp", "hp"),
                 c(Mean = "mean", SD = "sd"))

mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_at(c("mpg", "disp", "hp"),
                 funs(Mean = mean, SD = sd))


# A tibble: 3 x 7
#    cyl mpg_Mean disp_Mean   hp_Mean   mpg_SD  disp_SD    hp_SD
#  <dbl>    <dbl>     <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
#1     4 26.66364  105.1364  82.63636 4.509828 26.87159 20.93453
#2     6 19.74286  183.3143 122.28571 1.453567 41.56246 24.26049
#3     8 15.10000  353.1000 209.21429 2.560048 67.77132 50.97689
```

To select columns conditionally, use `summarise_if`:

Take the `mean` of all columns that are `numeric` grouped by `cyl`:

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_if(is.numeric, mean)

# A tibble: 3 x 11
#    cyl      mpg     disp        hp     drat       wt     qsec
#  <dbl>    <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
#1     4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727
#2     6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714
#3     8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214
# ... with 4 more variables: vs <dbl>, am <dbl>, gear <dbl>,
#   carb <dbl>
```

However, some variables are discrete, and `mean` of these variables doesn't make sense.

To take the `mean` of only continuous variables by `cyl`:

```
mtcars_tbl %>%
    group_by(cyl) %>%
    summarise_if(function(x) is.numeric(x) & n_distinct(x) > 6, mean)

# A tibble: 3 x 7
```

```
#    cyl      mpg     disp        hp     drat       wt     qsec
#  <dbl>    <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
#1     4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727
#2     6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714
#3     8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214
```

Subset Observation (Rows)

`dplyr::filter()` - Select a subset of rows in a data frame that meet a logical criteria:

```
dplyr::filter(iris,Sepal.Length>7)
#       Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
#    1           7.1         3.0          5.9         2.1 virginica
#    2           7.6         3.0          6.6         2.1 virginica
#    3           7.3         2.9          6.3         1.8 virginica
#    4           7.2         3.6          6.1         2.5 virginica
#    5           7.7         3.8          6.7         2.2 virginica
#    6           7.7         2.6          6.9         2.3 virginica
#    7           7.7         2.8          6.7         2.0 virginica
#    8           7.2         3.2          6.0         1.8 virginica
#    9           7.2         3.0          5.8         1.6 virginica
#    10          7.4         2.8          6.1         1.9 virginica
#    11          7.9         3.8          6.4         2.0 virginica
#    12          7.7         3.0          6.1         2.3 virginica
```

`dplyr::distinct()` - Remove duplicate rows:

```
distinct(iris, Sepal.Length, .keep_all = TRUE)
#       Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
#    1           5.1         3.5          1.4         0.2     setosa
#    2           4.9         3.0          1.4         0.2     setosa
#    3           4.7         3.2          1.3         0.2     setosa
#    4           4.6         3.1          1.5         0.2     setosa
#    5           5.0         3.6          1.4         0.2     setosa
#    6           5.4         3.9          1.7         0.4     setosa
#    7           4.4         2.9          1.4         0.2     setosa
#    8           4.8         3.4          1.6         0.2     setosa
#    9           4.3         3.0          1.1         0.1     setosa
#   10          5.8         4.0          1.2         0.2     setosa
#   11          5.7         4.4          1.5         0.4     setosa
#   12          5.2         3.5          1.5         0.2     setosa
#   13          5.5         4.2          1.4         0.2     setosa
#   14          4.5         2.3          1.3         0.3     setosa
#   15          5.3         3.7          1.5         0.2     setosa
#   16          7.0         3.2          4.7         1.4 versicolor
#   17          6.4         3.2          4.5         1.5 versicolor
#   18          6.9         3.1          4.9         1.5 versicolor
#   19          6.5         2.8          4.6         1.5 versicolor
#   20          6.3         3.3          4.7         1.6 versicolor
#   21          6.6         2.9          4.6         1.3 versicolor
#   22          5.9         3.0          4.2         1.5 versicolor
#   23          6.0         2.2          4.0         1.0 versicolor
#   24          6.1         2.9          4.7         1.4 versicolor
#   25          5.6         2.9          3.6         1.3 versicolor
```


www.dbooks.org

```
#   26          6.7         3.1          4.4         1.4 versicolor
#   27          6.2         2.2          4.5         1.5 versicolor
#   28          6.8         2.8          4.8         1.4 versicolor
#   29          7.1         3.0          5.9         2.1  virginica
#   30          7.6         3.0          6.6         2.1  virginica
#   31          7.3         2.9          6.3         1.8  virginica
#   32          7.2         3.6          6.1         2.5  virginica
#   33          7.7         3.8          6.7         2.2  virginica
#   34          7.4         2.8          6.1         1.9  virginica
#   35          7.9         3.8          6.4         2.0  virginica
```

Aggregating with %>% (pipe) operator

The pipe (%>%) operator could be used in combination with `dplyr` functions. In this example we use the `mtcars` dataset (see `help("mtcars")` for more information) to show how to sumarize a data frame, and to add variables to the data with the result of the application of a function.

```
library(dplyr)
library(magrittr)
df <- mtcars
df$cars <- rownames(df) #just add the cars names to the df
df <- df[,c(ncol(df),1:(ncol(df)-1))] # and place the names in the first column
```

1. Sumarize the data

To compute statistics we use `summarize` and the appropriate functions. In this case `n()` is used for counting the number of cases.

```
 df %>%
  summarize(count=n(),mean_mpg = mean(mpg, na.rm = TRUE),
            min_weight = min(wt),max_weight = max(wt))

#  count mean_mpg min_weight max_weight
#1    32 20.09062      1.513      5.424
```

2. Compute statistics by group

It is possible to compute the statistics by groups of the data. In this case by *Number of cylinders* and *Number of forward gears*

```
df %>%
  group_by(cyl, gear) %>%
  summarize(count=n(),mean_mpg = mean(mpg, na.rm = TRUE),
            min_weight = min(wt),max_weight = max(wt))

# Source: local data frame [8 x 6]
# Groups: cyl [?]
#
#    cyl  gear count mean_mpg min_weight max_weight
#  <dbl> <dbl> <int>    <dbl>      <dbl>      <dbl>
#1     4     3     1   21.500      2.465      2.465
#2     4     4     8   26.925      1.615      3.190
#3     4     5     2   28.200      1.513      2.140
#4     6     3     2   19.750      3.215      3.460
```

```
#5     6     4     4   19.750      2.620      3.440
#6     6     5     1   19.700      2.770      2.770
#7     8     3    12   15.050      3.435      5.424
#8     8     5     2   15.400      3.170      3.570
```

Examples of NSE and string variables in dpylr

`dplyr` uses Non-Standard Evaluation(NSE), which is why we normally can use the variable names without quotes. However, sometimes during the data pipeline, we need to get our variable names from other sources such as a Shiny selection box. In case of functions like `select`, we can just use `select_` to use a string variable to select

```
variable1 <- "Sepal.Length"
variable2 <- "Sepal.Width"
iris %>%
select_(variable1, variable2) %>%
head(n=5)
#  Sepal.Length Sepal.Width
#  1          5.1         3.5
#  2          4.9         3.0
#  3          4.7         3.2
#  4          4.6         3.1
#  5          5.0         3.6
```

But if we want to use other features such as summarize or filter we need to use `interp` function from `lazyeval` package

```
variable1 <- "Sepal.Length"
variable2 <- "Sepal.Width"
variable3 <- "Species"
iris %>%
select_(variable1, variable2, variable3) %>%
group_by_(variable3) %>%
summarize_(mean1 = lazyeval::interp(~mean(var), var = as.name(variable1)), mean2 =
lazyeval::interp(~mean(var), var = as.name(variable2)))
#      Species mean1 mean2
#       <fctr> <dbl> <dbl>
# 1     setosa 5.006 3.428
# 2 versicolor 5.936 2.770
# 3  virginica 6.588 2.974
```

Read dplyr online: https://riptutorial.com/r/topic/4250/dplyr

www.dbooks.org

Chapter 33: Expression: parse + eval

Remarks

The function `parse` convert text and files into expressions.

The function `eval` evaluate expressions.

Examples

Execute code in string format

In this exemple, we want to execute code which is stored in a string format.

```
# the string
str <- "1+1"

# A string is not an expression.
is.expression(str)
[1] FALSE

eval(str)
[1] "1+1"

# parse convert string into expressions
parsed.str <- parse(text="1+1")

is.expression(parsed.str)
[1] TRUE

eval(parsed.str)
[1] 2
```

Read Expression: parse + eval online: https://riptutorial.com/r/topic/5746/expression--parse-plus-eval

Chapter 34: Extracting and Listing Files in Compressed Archives

Examples

Extracting files from a .zip archive

Unzipping a zip archive is done with `unzip` function from the `utils` package (which is included in base R).

```
unzip(zipfile = "bar.zip", exdir = "./foo")
```

This will extract all files in `"bar.zip"` to the `"foo"` directory, which will be created if necessary. Tilde expansion is done automatically from your working directory. Alternatively, you can pass the whole path name to the zipfile.

Listing files in a .zip archive

Listing files in a zip archive is done with `unzip` function from the `utils` package (which is included in base R).

```
unzip(zipfile = "bar.zip", list = TRUE)
```

This will list all files in `"bar.zip"` and extract none. Tilde expansion is done automatically from your working directory. Alternatively, you can pass the whole path name to the zipfile.

Listing files in a .tar archive

Listing files in a tar archive is done with `untar` function from the `utils` package (which is included in base R).

```
untar(zipfile = "bar.tar", list = TRUE)
```

This will list all files in `"bar.tar"` and extract none. Tilde expansion is done automatically from your working directory. Alternatively, you can pass the whole path name to the tarfile.

Extracting files from a .tar archive

Extracting files from a tar archive is done with `untar` function from the `utils` package (which is included in base R).

```
untar(tarfile = "bar.tar", exdir = "./foo")
```

This will extract all files in `"bar.tar"` to the `"foo"` directory, which will be created if necessary. Tilde

www.dbooks.org

expansion is done automatically from your working directory. Alternatively, you can pass the whole path name to the tarfile.

Extract all .zip archives in a directory

With a simple `for` loop, all zip archives in a directory can be extracted.

```
for (i in dir(pattern=".zip$"))
    unzip(i)
```

The `dir` function produces a character vector of the names of the files in a directory matching the regex pattern specified by `pattern`. This vector is looped through with index `i`, using the `unzip` function to extract each zip archive.

Read Extracting and Listing Files in Compressed Archives online:
https://riptutorial.com/r/topic/4323/extracting-and-listing-files-in-compressed-archives

Chapter 35: Factors

Syntax

1. factor(x = character(), levels, labels = levels, exclude = NA, ordered = is.ordered(x), nmax = NA)
2. Run `?factor` or see the documentation online.

Remarks

An object with class `factor` is a vector with a particular set of characteristics.

1. It is stored internally as an `integer` vector.
2. It maintains a `levels` attribute the shows the character representation of the values.
3. Its class is stored as `factor`

To illustrate, let us generate a vector of 1,000 observations from a set of colors.

```
set.seed(1)
Color <- sample(x = c("Red", "Blue", "Green", "Yellow"),
                size = 1000,
                replace = TRUE)
Color <- factor(Color)
```

We can observe each of the characteristics of `Color` listed above:

```
#* 1. It is stored internally as an `integer` vector
typeof(Color)
```

```
[1] "integer"
```

```
#* 2. It maintains a `levels` attribute the shows the character representation of the values.
#* 3. Its class is stored as `factor`
attributes(Color)
```

```
$levels
[1] "Blue"   "Green"  "Red"    "Yellow"

$class
[1] "factor"
```

The primary advantage of a factor object is efficiency in data storage. An integer requires less memory to store than a character. Such efficiency was highly desirable when many computers had much more limited resources than current machines (for a more detailed history of the motivations behind using factors, see `stringsAsFactors`: an Unauthorized Biography). The difference in memory use can be seen even in our `Color` object. As you can see, storing `Color` as a character requires about 1.7 times as much memory as the factor object.

www.dbooks.org

```
#* Amount of memory required to store Color as a factor.
object.size(Color)
```

```
4624 bytes
```

```
#* Amount of memory required to store Color as a character
object.size(as.character(Color))
```

```
8232 bytes
```

Mapping the integer to the level

While the internal computation of factors sees the object as an integer, the desired representation for human consumption is the character level. For example,

```
head(Color)
```

```
[1] Blue   Blue   Green  Yellow Red    Yellow
Levels: Blue Green Red Yellow
```

is a easier for human comprehension than

```
head(as.numeric(Color))
```

```
[1] 1 1 2 4 3 4
```

An approximate illustration of how R goes about matching the character representation to the internal integer value is:

```
head(levels(Color)[as.numeric(Color)])
```

```
[1] "Blue"   "Blue"   "Green"  "Yellow" "Red"    "Yellow"
```

Compare these results to

```
head(Color)
```

```
[1] Blue   Blue   Green  Yellow Red    Yellow
Levels: Blue Green Red Yellow
```

Modern use of factors

In 2007, R introduced a hashing method for characters the reduced the memory burden of

character vectors (ref: `stringsAsFactors`: an Unauthorized Biography). Take note that when we determined that characters require 1.7 times more storage space than factors, that was calculated in a recent version of R, meaning that the memory use of character vectors was even more taxing before 2007.

Owing to the hashing method in modern R and to far greater memory resources in modern computers, the issue of memory efficiency in storing character values has been reduced to a very small concern. The prevailing attitude in the R Community is a preference for character vectors over factors in most situations. The primary causes for the shift away from factors are

1. The increase of unstructured and/or loosely controlled character data
2. The tendency of factors to not behave as desired when the user forgets she is dealing with a factor and not a character

In the first case, it makes no sense to store free text or open response fields as factors, as there will unlikely be any pattern that allows for more than one observation per level. Alternatively, if the data structure is not carefully controlled, it is possible to get multiple levels that correspond to the same category (such as "blue", "Blue", and "BLUE"). In such cases, many prefer to manage these discrepancies as characters prior to converting to a factor (if conversion takes place at all).

In the second case, if the user thinks she is working with a character vector, certain methods may not respond as anticipated. This basic understanding can lead to confusion and frustration while trying to debug scripts and codes. While, strictly speaking, this may be considered the fault of the user, most users are happy to avoid using factors and avoid these situations altogether.

Examples

Basic creation of factors

Factors are one way to represent categorical variables in R. A factor is stored internally as a **vector of integers**. The unique elements of the supplied character vector are known as the *levels* of the factor. By default, if the levels are not supplied by the user, then R will generate the set of unique values in the vector, sort these values alphanumerically, and use them as the levels.

```
 charvar <- rep(c("n", "c"), each = 3)
 f <- factor(charvar)
 f
 levels(f)

> f
[1] n n n c c c
Levels: c n
> levels(f)
[1] "c" "n"
```

If you want to change the ordering of the levels, then one option to to specify the levels manually:

```
levels(factor(charvar, levels = c("n","c")))

> levels(factor(charvar, levels = c("n","c")))
```


www.dbooks.org

```
[1] "n" "c"
```

Factors have a number of properties. For example, levels can be given labels:

```
> f <- factor(charvar, levels=c("n", "c"), labels=c("Newt", "Capybara"))
> f
[1] Newt     Newt     Newt     Capybara Capybara Capybara
Levels: Newt Capybara
```

Another property that can be assigned is whether the factor is ordered:

```
> Weekdays <- factor(c("Monday", "Wednesday", "Thursday", "Tuesday", "Friday", "Sunday",
"Saturday"))
> Weekdays
[1] Monday    Wednesday Thursday  Tuesday   Friday    Sunday    Saturday
Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday
> Weekdays <- factor(Weekdays, levels=c("Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday", "Sunday"), ordered=TRUE)
> Weekdays
[1] Monday    Wednesday Thursday  Tuesday   Friday    Sunday    Saturday
Levels: Monday < Tuesday < Wednesday < Thursday < Friday < Saturday < Sunday
```

When a level of the factor is no longer used, you can drop it using the `droplevels()` function:

```
> Weekend <- subset(Weekdays, Weekdays == "Saturday" |  Weekdays == "Sunday")
> Weekend
[1] Sunday   Saturday
Levels: Monday < Tuesday < Wednesday < Thursday < Friday < Saturday < Sunday
> Weekend <- droplevels(Weekend)
> Weekend
[1] Sunday   Saturday
Levels: Saturday < Sunday
```

Consolidating Factor Levels with a List

There are times in which it is desirable to consolidate factor levels into fewer groups, perhaps because of sparse data in one of the categories. It may also occur when you have varying spellings or capitalization of the category names. Consider as an example the factor

```
set.seed(1)
colorful <- sample(c("red", "Red", "RED", "blue", "Blue", "BLUE", "green", "gren"),
                   size = 20,
                   replace = TRUE)
colorful <- factor(colorful)
```

Since R is case-sensitive, a frequency table of this vector would appear as below.

```
table(colorful)
```

```
colorful
blue  Blue  BLUE green  gren   red   Red   RED
   3     1     4     2     4     1     3     2
```

This table, however, doesn't represent the true distribution of the data, and the categories may effectively be reduced to three types: Blue, Green, and Red. Three examples are provided. The first illustrates what seems like an obvious solution, but won't actually provide a solution. The second gives a working solution, but is verbose and computationally expensive. The third is not an obvious solution, but is relatively compact and computationally efficient.

Consolidating levels using `factor` (`factor_approach`)

```
factor(as.character(colorful),
       levels = c("blue", "Blue", "BLUE", "green", "gren", "red", "Red", "RED"),
       labels = c("Blue", "Blue", "Blue", "Green", "Green", "Red", "Red", "Red"))
```

```
 [1] Green Blue  Red   Red   Blue  Red   Red   Red   Blue  Red   Green Green Green
Blue  Red   Green
[17] Red   Green Green Red
Levels: Blue Blue Blue Green Green Red Red Red
Warning message:
In `levels<-`(`*tmp*`, value = if (nl == nL) as.character(labels) else
paste0(labels,  :
  duplicated levels in factors are deprecated
```

Notice that there are duplicated levels. We still have three categories for "Blue", which doesn't complete our task of consolidating levels. Additionally, there is a warning that duplicated levels are deprecated, meaning that this code may generate an error in the future.

Consolidating levels using `ifelse` (`ifelse_approach`)

```
factor(ifelse(colorful %in% c("blue", "Blue", "BLUE"),
       "Blue",
       ifelse(colorful %in% c("green", "gren"),
              "Green",
              "Red")))
```

```
 [1] Green Blue  Red   Red   Blue  Red   Red   Red   Blue  Red   Green Green Green
Blue  Red   Green
[17] Red   Green Green Red
Levels: Blue Green Red
```

This code generates the desired result, but requires the use of nested `ifelse` statements. While there is nothing wrong with this approach, managing nested `ifelse` statements can be a tedious task and must be done carefully.

Consolidating Factors Levels with a List (`list_approach`)

A less obvious way of consolidating levels is to use a list where the name of each element is the desired category name, and the element is a character vector of the levels in the factor that should map to the desired category. This has the added advantage of working directly on the `levels` attribute of the factor, without having to assign new objects.

www.dbooks.org

```
levels(colorful) <-
     list("Blue" = c("blue", "Blue", "BLUE"),
          "Green" = c("green", "gren"),
          "Red" = c("red", "Red", "RED"))
```

```
 [1] Green Blue  Red   Red   Blue  Red   Red   Red   Blue  Red   Green Green Green
Blue  Red   Green
[17] Red   Green Green Red
Levels: Blue Green Red
```

Benchmarking each approach

The time required to execute each of these approaches is summarized below. (For the sake of space, the code to generate this summary is not shown)

```
Unit: microseconds
          expr     min      lq      mean   median      uq     max neval cld
        factor  78.725  83.256  93.26023  87.5030  97.131 218.899   100  b
        ifelse 104.494 107.609 123.53793 113.4145 128.281 254.580   100   c
 list_approach  49.557  52.955  60.50756  54.9370  65.132 138.193   100 a
```

The list approach runs about twice as fast as the `ifelse` approach. However, except in times of very, very large amounts of data, the differences in execution time will likely be measured in either microseconds or milliseconds. With such small time differences, efficiency need not guide the decision of which approach to use. Instead, use an approach that is familiar and comfortable, and which you and your collaborators will understand on future review.

Factors

Factors are one method to represent categorical variables in R. Given a vector `x` whose values can be converted to characters using `as.character()`, the default arguments for `factor()` and `as.factor()` assign an integer to each distinct element of the vector as well as a level attribute and a label attribute. Levels are the values `x` can possibly take and labels can either be the given element or determined by the user.

To example how factors work we will create a factor with default attributes, then custom levels, and then custom levels and labels.

```
# standard
factor(c(1,1,2,2,3,3))
[1] 1 1 2 2 3 3
Levels: 1 2 3
```

Instances can arise where the user knows the number of possible values a factor can take on is greater than the current values in the vector. For this we assign the levels ourselves in `factor()`.

```
factor(c(1,1,2,2,3,3),
         levels = c(1,2,3,4,5))
[1] 1 1 2 2 3 3
Levels: 1 2 3 4 5
```

For style purposes the user may wish to assign labels to each level. By default, labels are the character representation of the levels. Here we assign labels for each of the possible levels in the factor.

```
factor(c(1,1,2,2,3,3),
       levels = c(1,2,3,4,5),
       labels = c("Fox","Dog","Cow","Brick","Dolphin"))
[1] Fox Fox Dog Dog Cow Cow
Levels: Fox Dog Cow Brick Dolphin
```

Normally, factors can only be compared using `==` and `!=` and if the factors have the same levels. The following comparison of factors fails even though they appear equal because the factors have different factor levels.

```
factor(c(1,1,2,2,3,3),levels = c(1,2,3)) == factor(c(1,1,2,2,3,3),levels = c(1,2,3,4,5))
Error in Ops.factor(factor(c(1, 1, 2, 2, 3, 3), levels = c(1, 2, 3)),  :
  level sets of factors are different
```

This makes sense as the extra levels in the RHS mean that R does not have enough information about each factor to compare them in a meaningful way.

The operators `<`, `<=`, `>` and `>=` are only usable for ordered factors. These can represent categorical values which still have a linear order. An ordered factor can be created by providing the `ordered = TRUE` argument to the `factor` function or just using the `ordered` function.

```
x <- factor(1:3, labels = c('low', 'medium', 'high'), ordered = TRUE)
print(x)
[1] low    medium high
Levels: low < medium < high

y <- ordered(3:1, labels = c('low', 'medium', 'high'))
print(y)
[1] high   medium low
Levels: low < medium < high

x < y
[1]  TRUE FALSE FALSE
```

For more information, see the Factor documentation.

Changing and reordering factors

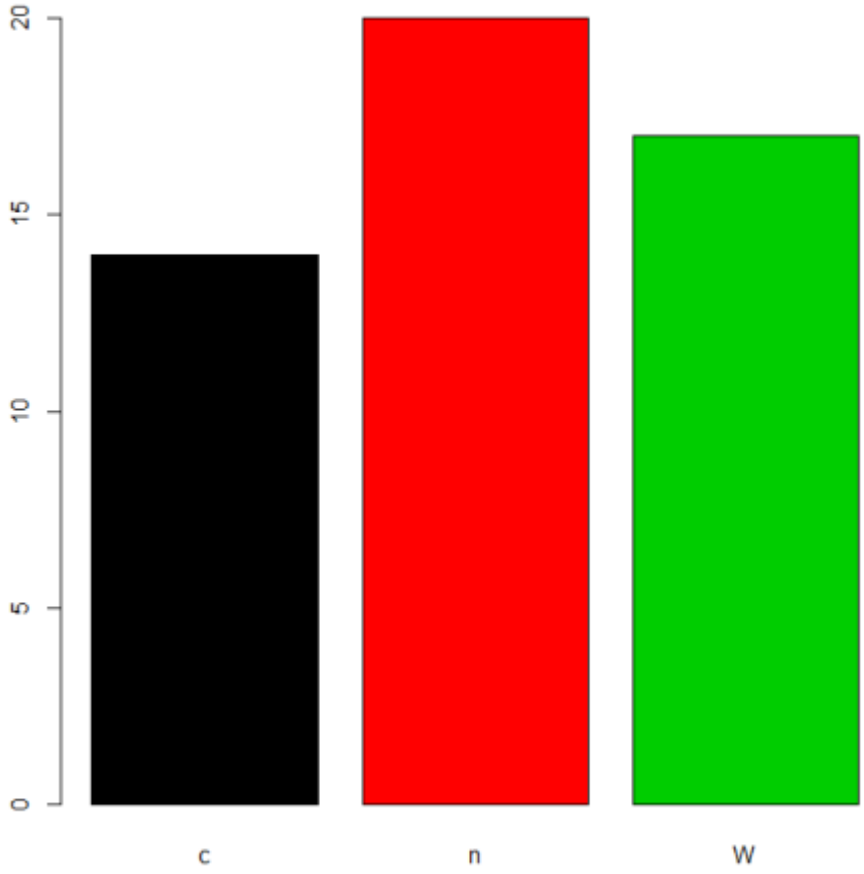
When factors are created with defaults, `levels` are formed by `as.character` applied to the inputs and are ordered alphabetically.

```
charvar <- rep(c("W", "n", "c"), times=c(17,20,14))
f <- factor(charvar)
levels(f)
# [1] "c" "n" "W"
```

In some situations the treatment of the default ordering of `levels` (alphabetic/lexical order) will be acceptable. For example, if one justs want to `plot` the frequencies, this will be the result:

www.dbooks.org

```
plot(f,col=1:length(levels(f)))
```



But if we want a different ordering of `levels`, we need to specify this in the `levels` or `labels` parameter (taking care that the meaning of "order" here is different from *ordered* factors, see below). There are many alternatives to accomplish that task depending on the situation.

1. Redefine the factor

When it is possible, we can recreate the factor using the `levels` parameter with the order we want.

```
ff <- factor(charvar, levels = c("n", "W", "c"))
levels(ff)
# [1] "n" "W" "c"

gg <- factor(charvar, levels = c("W", "c", "n"))
levels(gg)
# [1] "W" "c" "n"
```

When the input levels are different than the desired output levels, we use the `labels` parameter which causes the `levels` parameter to become a "filter" for acceptable input values, but leaves the final values of "levels" for the factor vector as the argument to `labels`:

```
fm <- factor(as.numeric(f),levels = c(2,3,1),
```

```
             labels = c("nn", "WW", "cc"))
levels(fm)
# [1] "nn" "WW" "cc"

fm <- factor(LETTERS[1:6], levels = LETTERS[1:4],  # only 'A'-'D' as input
                 labels = letters[1:4])            # but assigned to 'a'-'d'
fm
#[1] a    b    c    d    <NA> <NA>
#Levels: a b c d
```

2. Use `relevel` function

When there is one specific `level` that needs to be the first we can use `relevel`. This happens, for example, in the context of statistical analysis, when a `base` category is necessary for testing hypothesis.

```
g<-relevel(f, "n") # moves n to be the first level
levels(g)
# [1] "n" "c" "W"
```

As can be verified `f` and `g` are the same

```
all.equal(f, g)
# [1] "Attributes: < Component “levels”: 2 string mismatches >"
all.equal(f, g, check.attributes = F)
# [1] TRUE
```

3. Reordering factors

There are cases when we need to `reorder` the `levels` based on a number, a partial result, a computed statistic, or previous calculations. Let's reorder based on the **frequencies** of the `levels`

```
table(g)
# g
#  n  c  W
# 20 14 17
```

The `reorder` function is generic (see `help(reorder)`), but in this context needs: `x`, in this case the factor; `X`, a numeric value of the same length as `x`; and `FUN`, a function to be applied to `X` and computed by level of the `x`, which determines the `levels` order, by default increasing. The result is the same factor with its levels reordered.

```
g.ord <- reorder(g,rep(1,length(g)), FUN=sum) #increasing
levels(g.ord)
# [1] "c" "W" "n"
```

To get de decreasing order we consider negative values (`-1`)

```
g.ord.d <- reorder(g,rep(-1,length(g)), FUN=sum)
levels(g.ord.d)
# [1] "n" "W" "c"
```


www.dbooks.org

Again the factor is the same as the others.

```
data.frame(f,g,g.ord,g.ord.d)[seq(1,length(g),by=5),] #just same lines
#    f g g.ord g.ord.d
# 1  W W     W       W
# 6  W W     W       W
# 11 W W     W       W
# 16 W W     W       W
# 21 n n     n       n
# 26 n n     n       n
# 31 n n     n       n
# 36 n n     n       n
# 41 c c     c       c
# 46 c c     c       c
# 51 c c     c       c
```

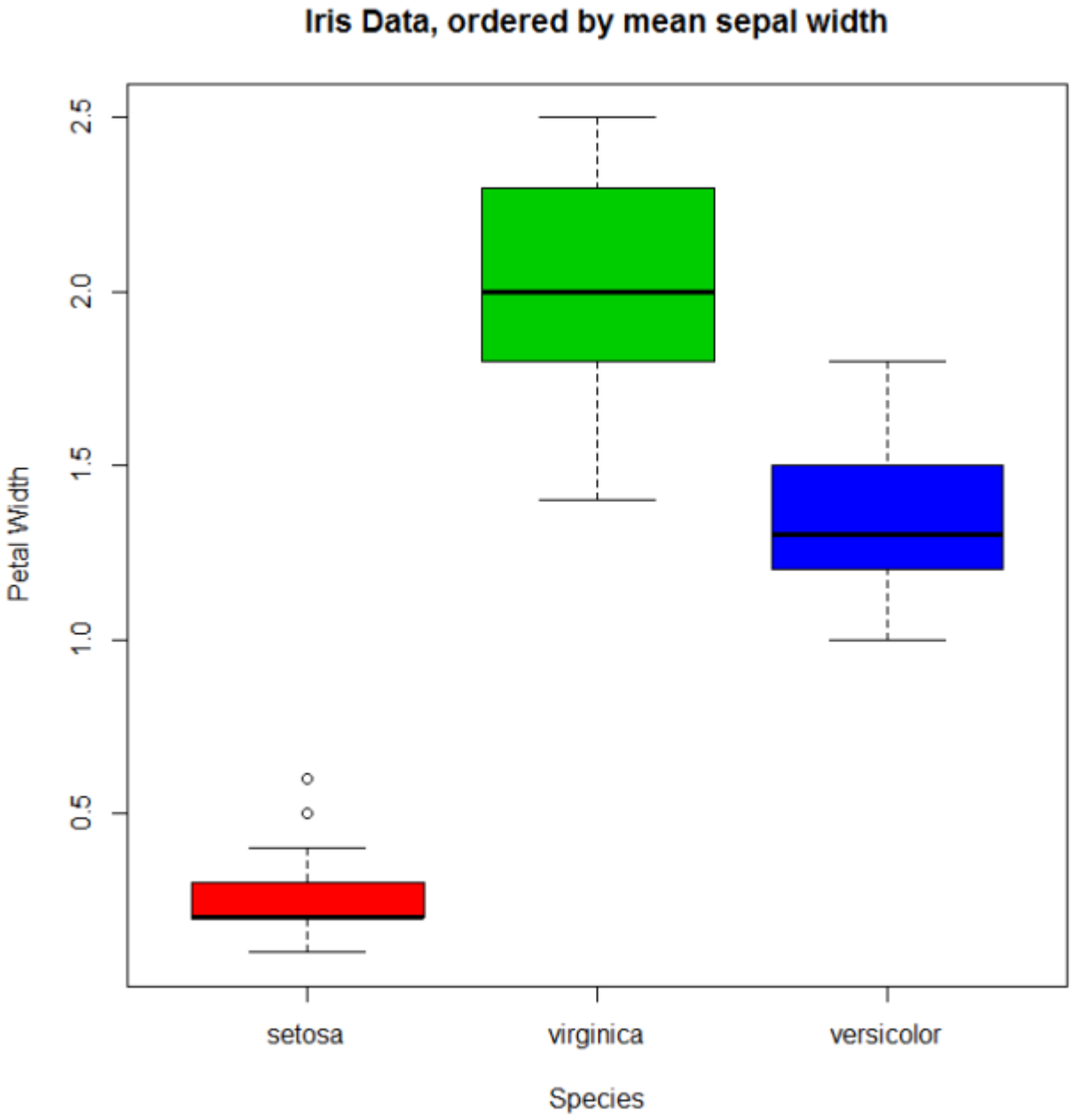
When there is a **quantitative variable** related to the factor variable, we could use other functions to reorder the `levels`. Lets take the `iris` data (`help("iris")` for more information), for reordering the `Species` factor by using its mean `Sepal.Width`.

```
miris <- iris  #help("iris") # copy the data
with(miris, tapply(Sepal.Width,Species,mean))
#    setosa versicolor  virginica
#     3.428      2.770      2.974

miris$Species.o<-with(miris,reorder(Species,-Sepal.Width))
levels(miris$Species.o)
# [1] "setosa"     "virginica"  "versicolor"
```

The usual `boxplot` (say: `with(miris, boxplot(Petal.Width~Species)`) will show the especies in this order: *setosa*, *versicolor*, and *virginica*. But using the ordered factor we get the species ordered by its mean `Sepal.Width`:

```
boxplot(Petal.Width~Species.o, data = miris,
        xlab = "Species", ylab = "Petal Width",
        main = "Iris Data, ordered by mean sepal width", varwidth = TRUE,
        col = 2:4)
```



Additionally, it is also possible to change the names of `levels`, combine them into groups, or add new `levels`. For that we use the function of the same name `levels`.

```
f1<-f
levels(f1)
# [1] "c" "n" "W"
levels(f1) <- c("upper","upper","CAP") #rename and grouping
levels(f1)
# [1] "upper" "CAP"

f2<-f1
levels(f2) <- c("upper","CAP", "Number") #add Number level, which is empty
levels(f2)
# [1] "upper"  "CAP"    "Number"
f2[length(f2):(length(f2)+5)]<-"Number" # add cases for the new level
table(f2)
# f2
```


www.dbooks.org

```
#  upper    CAP Number
#     33     17      6

f3<-f1
levels(f3) <- list(G1 = "upper", G2 = "CAP", G3 = "Number") # The same using list
levels(f3)
# [1] "G1" "G2" "G3"
f3[length(f3):(length(f3)+6)]<-"G3" ## add cases for the new level
table(f3)
# f3
# G1 G2 G3
# 33 17  7
```

- Ordered factors

Finally, we know that `ordered` factors are different from `factors`, the first one are used to represent *ordinal data*, and the second one to work with *nominal data*. At first, it does not make sense to change the order of `levels` for ordered factors, but we can change its `labels`.

```
ordvar<-rep(c("Low", "Medium", "High"), times=c(7,2,4))

of<-ordered(ordvar,levels=c("Low", "Medium", "High"))
levels(of)
# [1] "Low"    "Medium" "High"

of1<-of
levels(of1)<- c("LOW", "MEDIUM", "HIGH")
levels(of1)
# [1] "LOW"    "MEDIUM" "HIGH"
is.ordered(of1)
# [1] TRUE
of1
# [1] LOW    LOW    LOW    LOW    LOW    LOW    LOW    MEDIUM MEDIUM HIGH   HIGH   HIGH   HIGH

# Levels: LOW < MEDIUM < HIGH
```

Rebuilding factors from zero

Problem

Factors are used to represent variables that take values from a set of categories, known as Levels in R. For example, some experiment could be characterized by the energy level of a battery, with four levels: empty, low, normal, and full. Then, for 5 different sampling sites, those levels could be identified, in those terms, as follows:

> **full**, **full**, **normal**, **empty**, **low**

Typically, in databases or other information sources, the handling of these data is by arbitrary integer indices associated with the categories or levels. If we assume that, for the given example, we would assign, the indices as follows: 1 = empty, 2 = low, 3 = normal, 4 = full, then the 5 samples could be coded as:

> **4**, **4**, **3**, **1**, **2**

It could happen that, from your source of information, e.g. a database, you only have the encoded list of integers, and the catalog associating each integer with each level-keyword. How can a factor of R be reconstructed from that information?

Solution

We will simulate a vector of 20 integers that represents the samples, each of which may have one of four different values:

```
set.seed(18)
ii <- sample(1:4, 20, replace=T)
ii
```

```
[1] 4 3 4 1 1 3 2 3 2 1 3 4 1 2 4 1 3 1 4 1
```

The first step is to make a factor, from the previous sequence, in which the levels or categories are exactly the numbers from 1 to 4.

```
fii <- factor(ii, levels=1:4) # it is necessary to indicate the numeric levels
fii
```

```
[1] 4 3 4 1 1 3 2 3 2 1 3 4 1 2 4 1 3 1 4 1
Levels: 1 2 3 4
```

Now simply, you have to *dress* the factor already created with the index tags:

```
levels(fii) <- c("empty", "low", "normal", "full")
fii
```

```
[1] full normal full empty empty normal low normal low empty
[11] normal full empty low full empty normal empty full empty
Levels: empty low normal full
```

Read Factors online: https://riptutorial.com/r/topic/1104/factors

www.dbooks.org

Chapter 36: Fault-tolerant/resilient code

Parameters

Parameter	Details
expr	In case the "try part" was completed successfully `tryCatch` will return the **last evaluated expression**. Hence, the actual value being returned in case everything went well and there is no condition (i.e. a *warning* or an *error*) is the return value of `readLines`. Note that you don't need to explicilty state the return value via `return` as code in the "try part" is not wrapped insided a function environment (unlike that for the condition handlers for warnings and error below)
warning/error/etc	Provide/define a handler function for all the conditions that you want to handle explicitly. AFAIU, you can provide handlers for *any* type of conditions (not just *warnings* and *errors*, but also *custom* conditions; see `simpleCondition` and friends for that) as long as the **name of the respective handler function matches the class of the respective condition** (see the *Details* part of the doc for `tryCatch`).
finally	Here goes everything that should be executed at the very end, **regardless** if the expression in the "try part" succeeded or if there was any condition. If you want more than one expression to be executed, then you need to wrap them in curly brackets, otherwise you could just have written `finally = <expression>` (i.e. the same logic as for "try part".

Remarks

`tryCatch`

`tryCatch` returns the value associated to executing `expr` unless there's a condition: a warning or an error. If that's the case, specific return values (e.g. `return(NA)` above) can be specified by supplying a handler function for the respective conditions (see arguments `warning` and `error` in `?tryCatch`). These can be functions that already exist, but you can also define them within `tryCatch` (as we did above).

Implications of choosing specific return values of the handler functions

As we've specified that `NA` should be returned in case of an error in the "try part", the third element in `y` is `NA`. If we'd have chosen `NULL` to be the return value, the length of `y` would just have been `2` instead of `3` as `lapply` will simply "ignore/drop" return values that are `NULL`. Also note that if you don't specify an **explicit** return value via `return`, the handler functions will return `NULL` (i.e. in case of an *error* or a *warning* condition).

"Undesired" warning message

When the third element of our `urls` vector hits our function, we get the following warning **in addition** to the fact that an error occurs (`readLines` first complains that it can't open the connection via a *warning* before actually failing with an *error*):

```
Warning message:
    In file(con, "r") : cannot open file 'I'm no URL': No such file or directory
```

An *error* "wins" over a *warning*, so we're not really interested in the warning in this particular case. Thus we have set `warn = FALSE` in `readLines`, but that doesn't seem to have any effect. An alternative way to suppress the warning is to use

```
suppressWarnings(readLines(con = url))
```

instead of

```
readLines(con = url, warn = FALSE)
```

Examples

Using tryCatch()

We're defining a robust version of a function that reads the HTML code from a given URL. *Robust* in the sense that we want it to handle situations where something either goes wrong (error) or not quite the way we planned it to (warning). The umbrella term for errors and warnings is *condition*

Function definition using `tryCatch`

```
readUrl <- function(url) {
    out <- tryCatch(

        ########################################################
        # Try part: define the expression(s) you want to "try" #
        ########################################################

        {
            # Just to highlight:
            # If you want to use more than one R expression in the "try part"
            # then you'll have to use curly brackets.
            # Otherwise, just write the single expression you want to try and

            message("This is the 'try' part")
            readLines(con = url, warn = FALSE)
        },

        ########################################################################
        # Condition handler part: define how you want conditions to be handled #
        ########################################################################
```


www.dbooks.org

```
        # Handler when a warning occurs:
        warning = function(cond) {
            message(paste("Reading the URL caused a warning:", url))
            message("Here's the original warning message:")
            message(cond)

            # Choose a return value when such a type of condition occurs
            return(NULL)
        },

        # Handler when an error occurs:
        error = function(cond) {
            message(paste("This seems to be an invalid URL:", url))
            message("Here's the original error message:")
            message(cond)

            # Choose a return value when such a type of condition occurs
            return(NA)
        },

        ################################################
        # Final part: define what should happen AFTER #
        # everything has been tried and/or handled    #
        ################################################

        finally = {
            message(paste("Processed URL:", url))
            message("Some message at the end\n")
        }
    )
    return(out)
}
```

Testing things out

Let's define a vector of URLs where one element isn't a valid URL

```
urls <- c(
    "http://stat.ethz.ch/R-manual/R-devel/library/base/html/connections.html",
    "http://en.wikipedia.org/wiki/Xz",
    "I'm no URL"
)
```

And pass this as input to the function we defined above

```
y <- lapply(urls, readUrl)
# Processed URL: http://stat.ethz.ch/R-manual/R-devel/library/base/html/connections.html
# Some message at the end
#
# Processed URL: http://en.wikipedia.org/wiki/Xz
# Some message at the end
#
# URL does not seem to exist: I'm no URL
# Here's the original error message:
# cannot open the connection
# Processed URL: I'm no URL
# Some message at the end
```

```
#
# Warning message:
# In file(con, "r") : cannot open file 'I'm no URL': No such file or directory
```

Investigating the output

```
length(y)
# [1] 3

head(y[[1]])
# [1] "<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\">"
# [2] "<html><head><title>R: Functions to Manipulate Connections</title>"
# [3] "<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\">"
# [4] "<link rel=\"stylesheet\" type=\"text/css\" href=\"R.css\">"
# [5] "</head><body>"
# [6] ""

y[[3]]
# [1] NA
```

Read Fault-tolerant/resilient code online: https://riptutorial.com/r/topic/4060/fault-tolerant-resilient-code

www.dbooks.org

Chapter 37: Feature Selection in R -- Removing Extraneous Features

Examples

Removing features with zero or near-zero variance

A feature that has near zero variance is a good candidate for removal.

You can manually detect numerical variance below your own threshold:

```
data("GermanCredit")
variances<-apply(GermanCredit, 2, var)
variances[which(variances<=0.0025)]
```

Or, you can use the caret package to find near zero variance. An advantage here is that is defines near zero variance not in the numerical calculation of variance, but rather as a function of rarity:

> "nearZeroVar diagnoses predictors that have one unique value (i.e. are zero variance predictors) or predictors that are have both of the following characteristics: they have very few unique values relative to the number of samples and the ratio of the frequency of the most common value to the frequency of the second most common value is large..."

```
library(caret)
names(GermanCredit)[nearZeroVar(GermanCredit)]
```

Removing features with high numbers of NA

If a feature is largely lacking data, it is a good candidate for removal:

```
library(VIM)
data(sleep)
colMeans(is.na(sleep))

   BodyWgt   BrainWgt       NonD      Dream      Sleep       Span       Gest
0.00000000 0.00000000 0.22580645 0.19354839 0.06451613 0.06451613 0.06451613
      Pred        Exp     Danger
0.00000000 0.00000000 0.00000000
```

In this case, we may want to remove NonD and Dream, which each have around 20% missing values (your cutoff may vary)

Removing closely correlated features

Closely correlated features may add variance to your model, and removing one of a correlated pair

might help reduce that. There are lots of ways to detect correlation. Here's one:

```
library(purrr) # in order to use keep()

# select correlatable vars
toCorrelate<-mtcars %>% keep(is.numeric)

# calculate correlation matrix
correlationMatrix <- cor(toCorrelate)

# pick only one out of each highly correlated pair's mirror image
correlationMatrix[upper.tri(correlationMatrix)]<-0

# and I don't remove the highly-correlated-with-itself group
diag(correlationMatrix)<-0

# find features that are highly correlated with another feature at the +- 0.85 level
apply(correlationMatrix,2, function(x) any(abs(x)>=0.85))

  mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
 TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

I'll want to look at what MPG is correlated to so strongly, and decide what to keep and what to toss. Same for cyl and disp. Alternatively, I might need to combine some strongly correlated features.

Read Feature Selection in R -- Removing Extraneous Features online:
https://riptutorial.com/r/topic/7561/feature-selection-in-r----removing-extraneous-features

www.dbooks.org

Chapter 38: Formula

Examples

The basics of formula

Statistical functions in R make heavy use of the so-called Wilkinson-Rogers formula notation[1] .

When running model functions like `lm` for the Linear Regressions, they need a `formula`. This `formula` specifies which regression coefficients shall be estimated.

```
my_formula1 <- formula(mpg ~ wt)
class(my_formula1)
# gives "formula"

mod1 <- lm(my_formula1, data = mtcars)
coef(mod1)
# gives (Intercept)          wt
#         37.285126   -5.344472
```

On the left side of the `~` (LHS) the dependent variable is specified, while the right hand side (RHS) contains the independent variables. Technically the `formula` call above is redundant because the tilde-operator is an infix function that returns an object with formula class:

```
form <- mpg ~ wt
class(form)
#[1] "formula"
```

The advantage of the `formula` function over `~` is that it also allows an environment for evaluation to be specified:

```
form_mt <- formula(mpg ~ wt, env = mtcars)
```

In this case, the output shows that a regression coefficient for `wt` is estimated, as well as (per default) an intercept parameter. The intercept can be excluded / forced to be 0 by including `0` or `-1` in the `formula`:

```
coef(lm(mpg ~ 0 + wt, data = mtcars))
coef(lm(mpg ~ wt -1, data = mtcars))
```

Interactions between variables `a` and `b` can added by included `a:b` to the `formula`:

```
 coef(lm(mpg ~ wt:vs, data = mtcars))
```

As it is (from a statistical point of view) generally advisable not have interactions in the model without the main effects, the naive approach would be to expand the `formula` to `a + b + a:b`. This works but can be simplified by writing `a*b`, where the `*` operator indicates factor crossing (when

between two factor columns) or multiplication when one or both of the columns are 'numeric':

```
coef(lm(mpg ~ wt*vs, data = mtcars))
```

Using the `*` notation expands a term to include all lower order effects, such that:

```
coef(lm(mpg ~ wt*vs*hp, data = mtcars))
```

will give, in addition to the intercept, 7 regression coefficients. One for the three-way interaction, three for the two-way interactions and three for the main effects.

If one wants, for example, to exclude the three-way interaction, but retain all two-way interactions there are two shorthands. First, using `-` we can subtract any particular term:

```
coef(lm(mpg ~ wt*vs*hp - wt:vs:hp, data = mtcars))
```

Or, we can use the `^` notation to specify which level of interaction we require:

```
coef(lm(mpg ~ (wt + vs + hp) ^ 2, data = mtcars))
```

Those two formula specifications should create the same model matrix.

Finally, `.` is shorthand to use all available variables as main effects. In this case, the `data` argument is used to obtain the available variables (which are not on the LHS). Therefore:

```
coef(lm(mpg ~ ., data = mtcars))
```

gives coefficients for the intercept and 10 independent variables. This notation is frequently used in machine learning packages, where one would like to use all variables for prediction or classification. Note that the meaning of `.` depends on context (see e.g. `?update.formula` for a different meaning).

1. G. N. Wilkinson and C. E. Rogers. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* Vol. 22, No. 3 (1973), pp. 392-399

Create Linear, Quadratic and Second Order Interaction Terms

`y ~ .` : Here `.` is interpreted as all variables except `y` in the data frame used in fitting the model. It is equivalent to the linear combinations of predictor variables. For example `y ~ var1 + var2 + var3+...+var15`

`y ~ . ^ 2` will give all linear (main effects) and second order interaction terms of the variables in the data frame. It is equivalent to `y ~ var1 + var2 + ...+var15 + var1:var2 + var1:var3 + var1:var4...and so on`

`y ~ var1 + var2 + ...+var15 + I(var1^2) + I(var2^2) + I(var3^2)...+I(var15^2)` : Here `I(var^2)` indicates quadratic polynomial of one variable in the data frame.

`y ~ poly(var1, degree = 2) + poly(var2, degree = 2)+...poly(var15, degree = 2)`

www.dbooks.org

or

`y ~ poly(var1, var2, var3, ....var15, degree = 2)` will be equivalent to the above expression.

`poly(var1, degree = 2)` is equivalent to `var1 + I(var1^2)`.

To get cubic polynomials, use `degree = 3` in `poly()`.

There is a caveat in using `poly` versus `I(var, 2)`, which is after fitting the model, each of them will produce different coefficients, but the fitted values are equivalent, because they represent different parameterizations of the same model. It is recommended to use `I(var, 2)` over `poly()` to avoid the summary effect seen in `poly()`.

In summary, to get linear, quadratic and second order interaction terms, you will have an expression like

`y ~ .^2 + I(var1^2) + I(var2^2)+...I(var15^2)`

Demo for four variables:

```
old <- reformulate( 'y ~ x1+x2+x3+x4' )
new <- reformulate( " y ~ .^2 + I(x1^2) + I(x2^2) + I(x3^2) + I(x4^2) ")
tmp <- .Call(stats:::C_updateform, old, new)
terms.formula(tmp, simplify = TRUE )

# ~y ~ x1 + x2 + x3 + x4 + I(x1^2) + I(x2^2) + I(x3^2) + I(x4^2) +
#   x1:x2 + x1:x3 + x1:x4 + x2:x3 + x2:x4 + x3:x4
# attr(,"variables")
# list(~y, x1, x2, x3, x4, I(x1^2), I(x2^2), I(x3^2), I(x4^2))
# attr(,"factors")
#         x1 x2 x3 x4 I(x1^2) I(x2^2) I(x3^2) I(x4^2) x1:x2 x1:x3 x1:x4 x2:x3 x2:x4 x3:x4
# ~y       0  0  0  0       0       0       0       0     0     0     0     0     0     0
# x1       1  0  0  0       0       0       0       0     1     1     1     0     0     0
# x2       0  1  0  0       0       0       0       0     1     0     0     1     1     0
# x3       0  0  1  0       0       0       0       0     0     1     0     1     0     1
# x4       0  0  0  1       0       0       0       0     0     0     1     0     1     1
# I(x1^2)  0  0  0  0       1       0       0       0     0     0     0     0     0     0
# I(x2^2)  0  0  0  0       0       1       0       0     0     0     0     0     0     0
# I(x3^2)  0  0  0  0       0       0       1       0     0     0     0     0     0     0
# I(x4^2)  0  0  0  0       0       0       0       1     0     0     0     0     0     0
# attr(,"term.labels")
# [1] "x1"      "x2"      "x3"      "x4"      "I(x1^2)" "I(x2^2)" "I(x3^2)" "I(x4^2)"
# [9] "x1:x2"   "x1:x3"   "x1:x4"   "x2:x3"   "x2:x4"   "x3:x4"
# attr(,"order")
# [1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2
# attr(,"intercept")
# [1] 1
# attr(,"response")
# [1] 1
# attr(,".Environment")
# <environment: R_GlobalEnv>
```

Read Formula online: https://riptutorial.com/r/topic/1061/formula

Chapter 39: Fourier Series and Transformations

Remarks

The Fourier transform decomposes a function of time (a signal) into the frequencies that make it up, similarly to how a musical chord can be expressed as the amplitude (or loudness) of its constituent notes. The Fourier transform of a function of time itself is a complex-valued function of frequency, whose absolute value represents the amount of that frequency present in the original function, and whose complex argument is the phase offset of the basic sinusoid in that frequency.

The Fourier transform is called the frequency domain representation of the original signal. The term Fourier transform refers to both the frequency domain representation and the mathematical operation that associates the frequency domain representation to a function of time. The Fourier transform is not limited to functions of time, but in order to have a unified language, the domain of the original function is commonly referred to as the time domain. For many functions of practical interest one can define an operation that reverses this: the inverse Fourier transformation, also called Fourier synthesis, of a frequency domain representation combines the contributions of all the different frequencies to recover the original function of time.

Linear operations performed in one domain (time or frequency) have corresponding operations in the other domain, which are sometimes easier to perform. The operation of differentiation in the time domain corresponds to multiplication by the frequency, so some differential equations are easier to analyze in the frequency domain. Also, convolution in the time domain corresponds to ordinary multiplication in the frequency domain. Concretely, this means that any linear time-invariant system, such as an electronic filter applied to a signal, can be expressed relatively simply as an operation on frequencies. So significant simplification is often achieved by transforming time functions to the frequency domain, performing the desired operations, and transforming the result back to time.

Harmonic analysis is the systematic study of the relationship between the frequency and time domains, including the kinds of functions or operations that are "simpler" in one or the other, and has deep connections to almost all areas of modern mathematics.

Functions that are localized in the time domain have Fourier transforms that are spread out across the frequency domain and vice versa. The critical case is the Gaussian function, of substantial importance in probability theory and statistics as well as in the study of physical phenomena exhibiting normal distribution (e.g., diffusion), which with appropriate normalizations goes to itself under the Fourier transform. Joseph Fourier introduced the transform in his study of heat transfer, where Gaussian functions appear as solutions of the heat equation.

The Fourier transform can be formally defined as an improper Riemann integral, making it an integral transform, although this definition is not suitable for many applications requiring a more sophisticated integration theory.

www.dbooks.org

For example, many relatively simple applications use the Dirac delta function, which can be treated formally as if it were a function, but the justification requires a mathematically more sophisticated viewpoint. The Fourier transform can also be generalized to functions of several variables on Euclidean space, sending a function of 3-dimensional space to a function of 3-dimensional momentum (or a function of space and time to a function of 4-momentum).

This idea makes the spatial Fourier transform very natural in the study of waves, as well as in quantum mechanics, where it is important to be able to represent wave solutions either as functions either of space or momentum and sometimes both. In general, functions to which Fourier methods are applicable are complex-valued, and possibly vector-valued. Still further generalization is possible to functions on groups, which, besides the original Fourier transform on $\mathbb{R}$ or $\mathbb{R}n$ (viewed as groups under addition), notably includes the discrete-time Fourier transform (DTFT, group = $\mathbb{Z}$), the discrete Fourier transform (DFT, group = $\mathbb{Z}$ mod N) and the Fourier series or circular Fourier transform (group = S1, the unit circle ≈ closed finite interval with endpoints identified). The latter is routinely employed to handle periodic functions. The Fast Fourier transform (FFT) is an algorithm for computing the DFT.

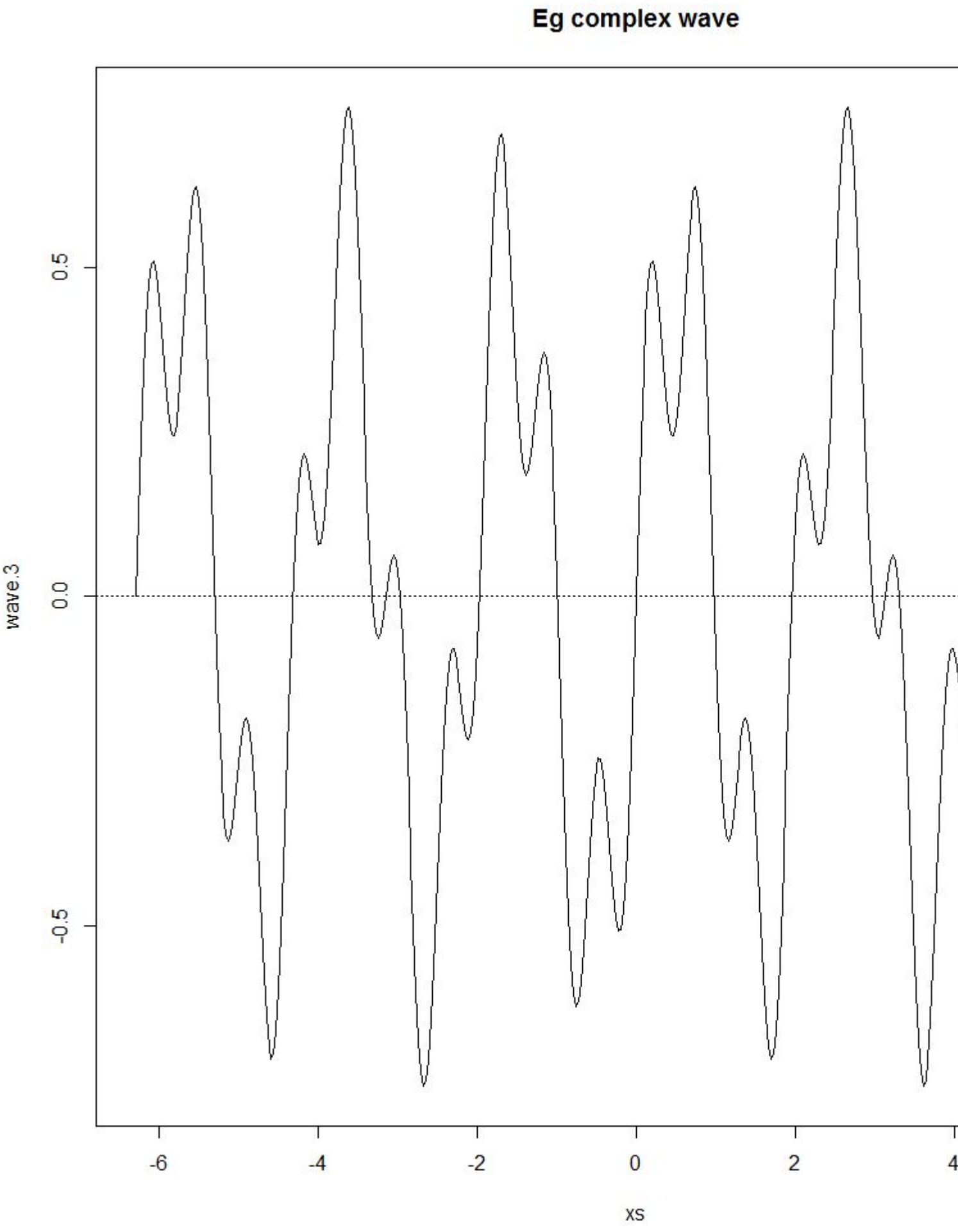
Examples

Fourier Series

Joseph Fourier showed that any periodic wave can be represented by a sum of simple sine waves. This sum is called the Fourier Series. The Fourier Series only holds while the system is linear. If there is, eg, some overflow effect (a threshold where the output remains the same no matter how much input is given), a non-linear effect enters the picture, breaking the sinusoidal wave and the superposition principle.

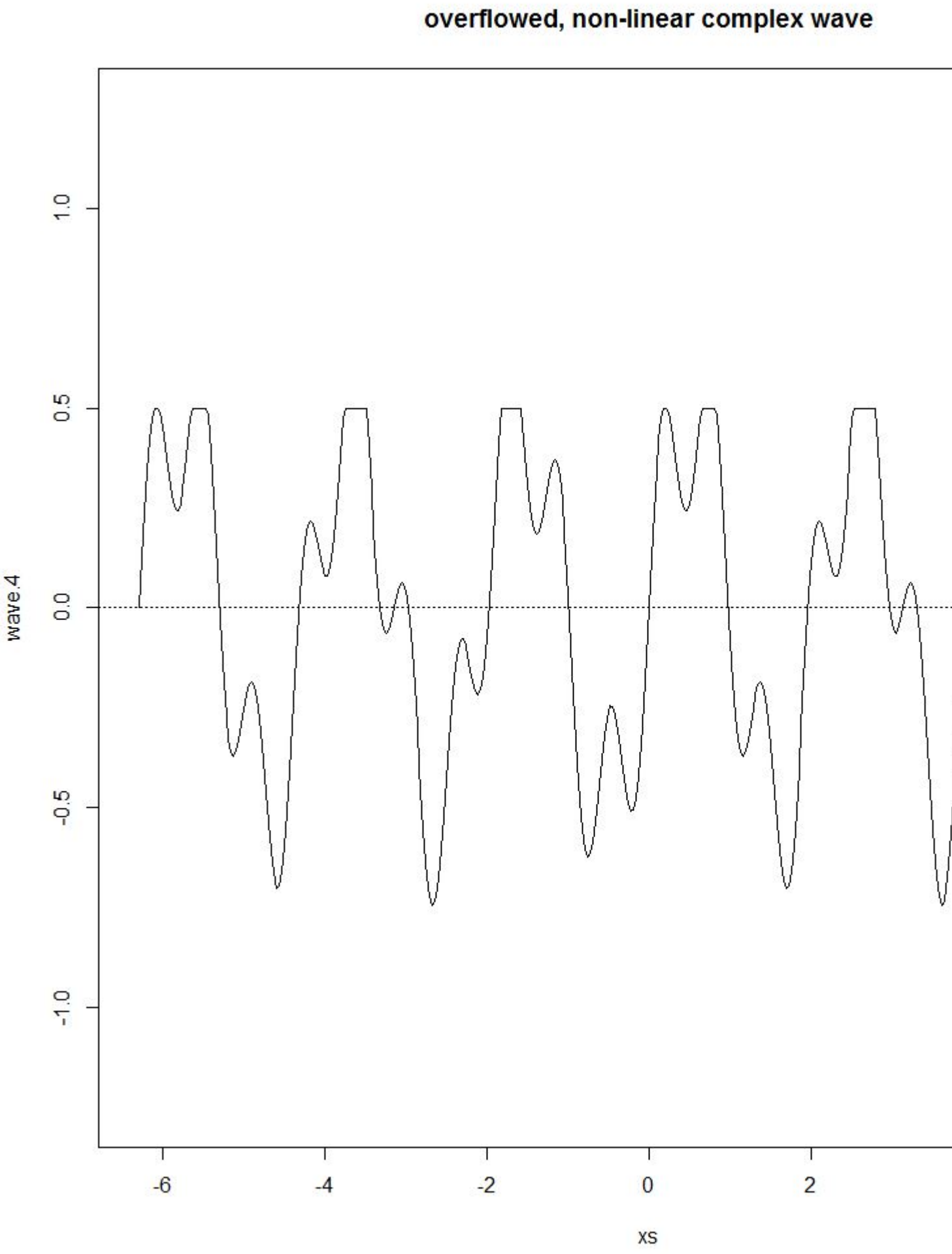
```
# Sine waves
xs <- seq(-2*pi,2*pi,pi/100)
wave.1 <- sin(3*xs)
wave.2 <- sin(10*xs)
par(mfrow = c(1, 2))
plot(xs,wave.1,type="l",ylim=c(-1,1)); abline(h=0,lty=3)
plot(xs,wave.2,type="l",ylim=c(-1,1)); abline(h=0,lty=3)

# Complex Wave
wave.3 <- 0.5 * wave.1 + 0.25 * wave.2
plot(xs,wave.3,type="l"); title("Eg complex wave"); abline(h=0,lty=3)
```



www.dbooks.org

```
wave.4 <- wave.3
wave.4[wave.3>0.5] <- 0.5
plot(xs,wave.4,type="l",ylim=c(-1.25,1.25))
title("overflowed, non-linear complex wave")
abline(h=0,lty=3)
```



www.dbooks.org

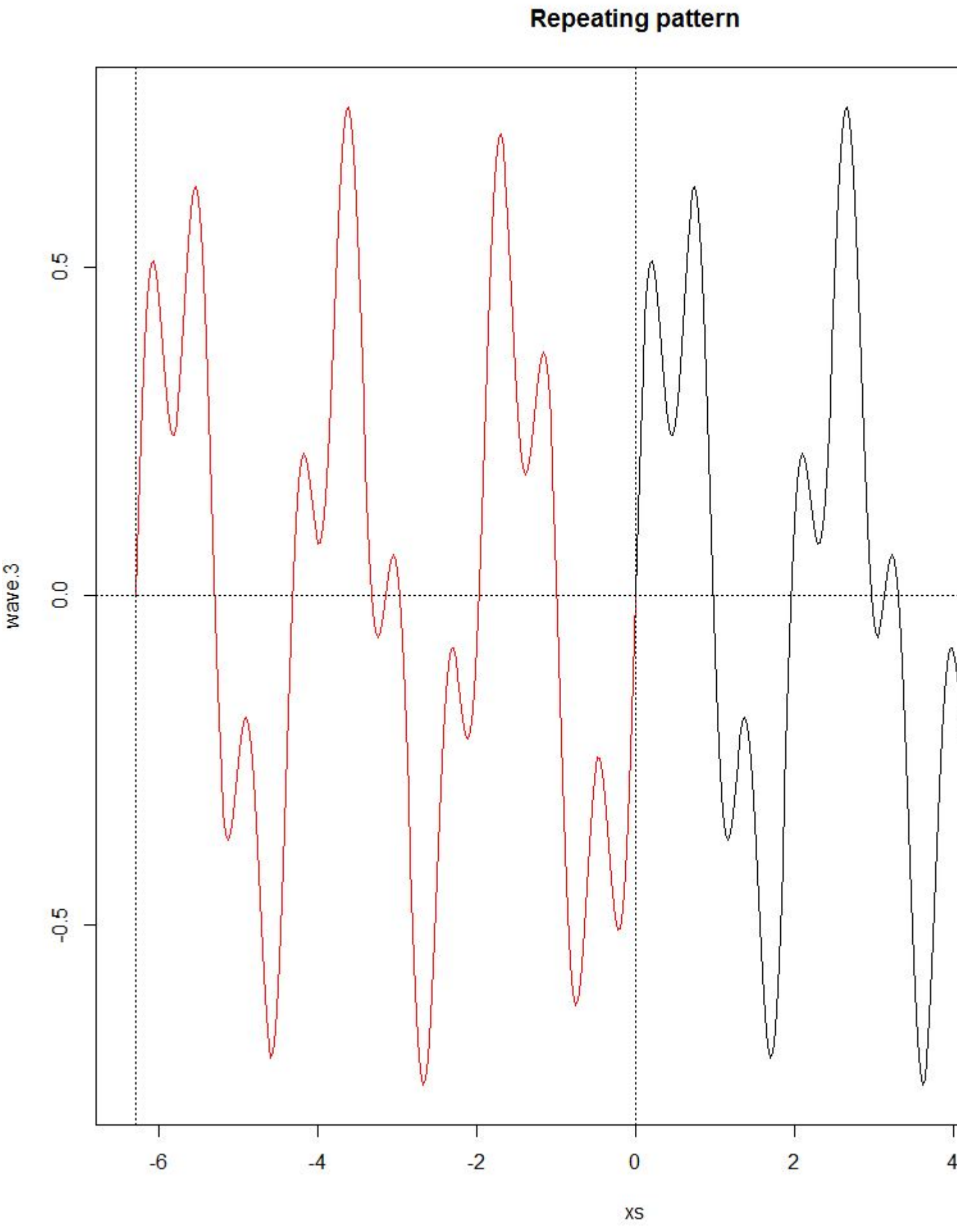
Also, the Fourier Series only holds if the waves are periodic, ie, they have a repeating pattern (non periodic waves are dealt by the Fourier Transform, see below). A periodic wave has a frequency f and a wavelength λ (a wavelength is the distance in the medium between the beginning and end of a cycle, λ=v/f0, where v is the wave velocity) that are defined by the repeating pattern. A non-periodic wave does not have a frequency or wavelength.

Some concepts:

- The fundamental period, T, is the period of all the samples taken, the time between the first sample and the last
- The sampling rate, sr, is the number of samples taken over a time period (aka acquisition frequency). For simplicity we will make the time interval between samples equal. This time interval is called the sample interval, si, which is the fundamental period time divided by the number of samples N. So, si=TN
- The fundamental frequency, f0, which is 1T. The fundamental frequency is the frequency of the repeating pattern or how long the wavelength is. In the previous waves, the fundamental frequency was 12π. The frequencies of the wave components must be integer multiples of the fundamental frequency. f0 is called the first harmonic, the second harmonic is 2∗f0, the third is 3∗f0, etc.

```
repeat.xs     <- seq(-2*pi,0,pi/100)
wave.3.repeat <- 0.5*sin(3*repeat.xs) + 0.25*sin(10*repeat.xs)
plot(xs,wave.3,type="l")

title("Repeating pattern")
points(repeat.xs,wave.3.repeat,type="l",col="red");
abline(h=0,v=c(-2*pi,0),lty=3)
```



www.dbooks.org

Here's a R function for plotting trajectories given a fourier series:

```
plot.fourier <- function(fourier.series, f.0, ts) {
                        w <- 2*pi*f.0 trajectory <- sapply(ts, function(t)
fourier.series(t,w))
                        plot(ts, trajectory, type="l", xlab="time", ylab="f(t)");
                        abline(h=0,lty=3)}
```

Read Fourier Series and Transformations online: https://riptutorial.com/r/topic/4139/fourier-series-and-transformations

Chapter 40: Functional programming

Examples

Built-in Higher Order Functions

R has a set of built in higher order functions: `Map`, `Reduce`, `Filter`, `Find`, `Position`, `Negate`.

`Map` applies a given function to a list of values:

```
words <- list("this", "is", "an", "example")
Map(toupper, words)
```

`Reduce` successively applies a binary function to a list of values in a recursive fashion.

```
Reduce(`*`, 1:10)
```

`Filter` given a predicate function and a list of values returns a filtered list containing only values for whom predicate function is TRUE.

```
Filter(is.character, list(1,"a",2,"b",3,"c"))
```

`Find` given a predicate function and a list of values returns the first value for which the predicate function is TRUE.

```
Find(is.character, list(1,"a",2,"b",3,"c"))
```

`Position` given a predicate function and a list of values returns the position of the first value in the list for which the predicate function is TRUE.

```
Position(is.character, list(1,"a",2,"b",3,"c"))
```

`Negate` inverts a predicate function making it return FALSE for values where it returned TRUE and vice versa.

```
is.noncharacter <- Negate(is.character)
is.noncharacter("a")
is.noncharacter(mean)
```

Read Functional programming online: https://riptutorial.com/r/topic/5050/functional-programming

www.dbooks.org

Chapter 41: Generalized linear models

Examples

Logistic regression on Titanic dataset

Logistic regression is a particular case of the *generalized linear model*, used to model dichotomous outcomes (*probit* and *complementary log-log* models are closely related).

The name comes from the *link function* used, the *logit* or *log-odds* function. The inverse function of the *logit* is called the *logistic function* and is given by:

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$$

This function takes a value between *]-Inf;+Inf[* and returns a value between *0* and *1*; i.e the *logistic function* takes a linear predictor and returns a probability.

Logistic regression can be performed using the `glm` function with the option `family = binomial` (shortcut for `family = binomial(link="logit")`; the *logit* being the default link function for the binomial family).

In this example, we try to predict the fate of the passengers aboard the RMS Titanic.

Read the data:

```
url <- "http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.txt"
titanic <- read.csv(file = url, stringsAsFactors = FALSE)
```

Clean the missing values:

In that case, we replace the missing values by an approximation, the average.

```
titanic$age[is.na(titanic$age)] <- mean(titanic$age, na.rm = TRUE)
```

Train the model:

```
titanic.train <- glm(survived ~ pclass + sex + age,
                         family = binomial, data = titanic)
```

Summary of the model:

```
summary(titanic.train)
```

The output:

```
Call:
```

```
glm(formula = survived ~ pclass + sex + age, family = binomial, data = titanic)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.6452  -0.6641  -0.3679   0.6123   2.5615

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.552261   0.342188  10.381  < 2e-16 ***
pclass2nd   -1.170777   0.211559  -5.534 3.13e-08 ***
pclass3rd   -2.430672   0.195157 -12.455  < 2e-16 ***
sexmale     -2.463377   0.154587 -15.935  < 2e-16 ***
age         -0.042235   0.007415  -5.696 1.23e-08 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 1686.8  on 1312  degrees of freedom
Residual deviance: 1165.7  on 1308  degrees of freedom
AIC: 1175.7

Number of Fisher Scoring iterations: 5
```

- The first thing displayed is the call. It is a reminder of the model and the options specified.
- Next we see the deviance residuals, which are a measure of model fit. This part of output shows the distribution of the deviance residuals for individual cases used in the model.
- The next part of the output shows the coefficients, their standard errors, the z-statistic (sometimes called a Wald z-statistic), and the associated p-values.
    - The qualitative variables are "dummified". A modality is considered as the reference. The reference modality can be change with `I` in the formula.
    - All four predictors are statistically significant at a 0.1 % level.
    - The logistic regression coefficients give the change in the log odds of the outcome for a one unit increase in the predictor variable.
    - To see the *odds ratio* (multiplicative change in the odds of survival per unit increase in a predictor variable), exponentiate the parameter.
    - To see the confidence interval (CI) of the parameter, use `confint`.
- Below the table of coefficients are fit indices, including the null and deviance residuals and the Akaike Information Criterion (AIC), which can be used for comparing model performance.
    - When comparing models fitted by maximum likelihood to the same data, the smaller the AIC, the better the fit.
    - One measure of model fit is the significance of the overall model. This test asks whether the model with predictors fits significantly better than a model with just an intercept (i.e., a null model).

Example of odds ratios:

```
exp(coef(titanic.train)[3])
```


www.dbooks.org

```
 pclass3rd
0.08797765
```

With this model, compared to the first class, the 3rd class passengers have about a tenth of the odds of survival.

Example of confidence interval for the parameters:

```
confint(titanic.train)

Waiting for profiling to be done...
                 2.5 %      97.5 %
(Intercept)  2.89486872  4.23734280
pclass2nd   -1.58986065 -0.75987230
pclass3rd   -2.81987935 -2.05419500
sexmale     -2.77180962 -2.16528316
age         -0.05695894 -0.02786211
```

Exemple of calculating the significance of the overall model:

The test statistic is distributed chi-squared with degrees of freedom equal to the differences in degrees of freedom between the current and the null model (i.e., the number of predictor variables in the model).

```
with(titanic.train, pchisq(null.deviance - deviance, df.null - df.residual
, lower.tail = FALSE))
[1] 1.892539e-111
```

The p-value is near 0, showing a strongly significant model.

Read Generalized linear models online: https://riptutorial.com/r/topic/2892/generalized-linear-models

Chapter 42: Get user input

Syntax

- variable <- readline(prompt = "Any message for user")
- name <- readline(prompt = "What's your name")

Examples

User input in R

Sometimes it can be interesting to have a cross-talk between the user and the program, one example being the swirl package that had been designed to teach R in R.

One can ask for user input using the `readline` command:

```
name <- readline(prompt = "What is your name?")
```

The user can then give any answer, such as a number, a character, vectors, and scanning the result is here to make sure that the user has given a proper answer. For example:

```
result <- readline(prompt = "What is the result of 1+1?")
while(result!=2){
    readline(prompt = "Wrong answer. What is the result of 1+1?")
}
```

However, it is to be noted that this code be stuck in a never-ending loop, as user input is saved as a character.

We have to coerce it to a number, using `as.numeric`:

```
result <- as.numeric(readline(prompt = "What is the result of 1+1?"))
while(result!=2){
    readline(prompt = "Wrong answer. What is the result of 1+1?")
}
```

Read Get user input online: https://riptutorial.com/r/topic/5098/get-user-input

www.dbooks.org

Chapter 43: ggplot2

Remarks

`ggplot2` has its own perfect reference website http://ggplot2.tidyverse.org/.

Most of the time, it is more convenient to adapt the structure or content of the plotted data (e.g. a `data.frame`) than adjusting things within the plot afterwards.

RStudio publishes a very helpful "Data Visualization with ggplot2" cheatsheet that can be found here.

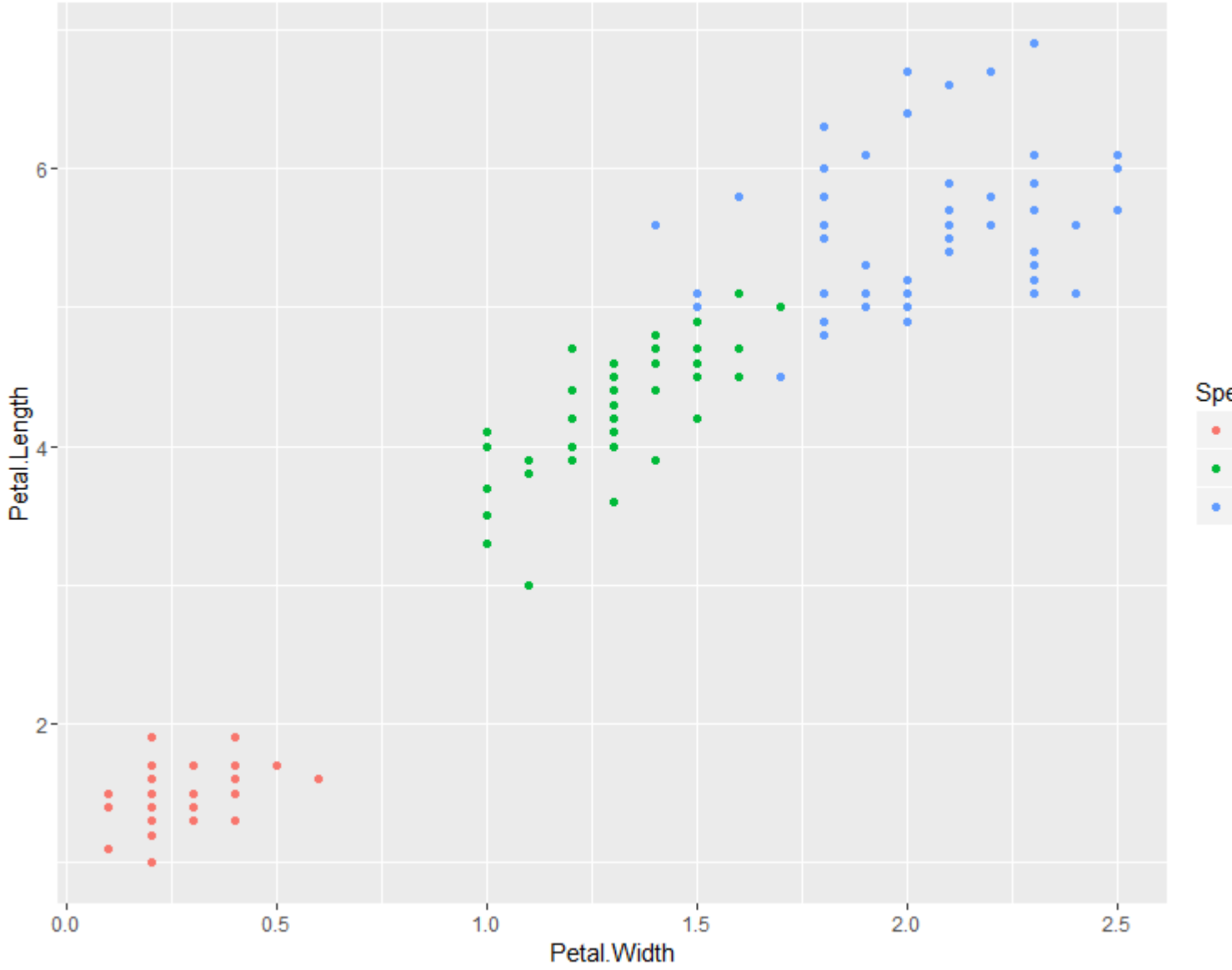
Examples

Scatter Plots

We plot a simple scatter plot using the builtin iris data set as follows:

```
library(ggplot2)
ggplot(iris, aes(x = Petal.Width, y = Petal.Length, color = Species)) +
  geom_point()
```

This gives:



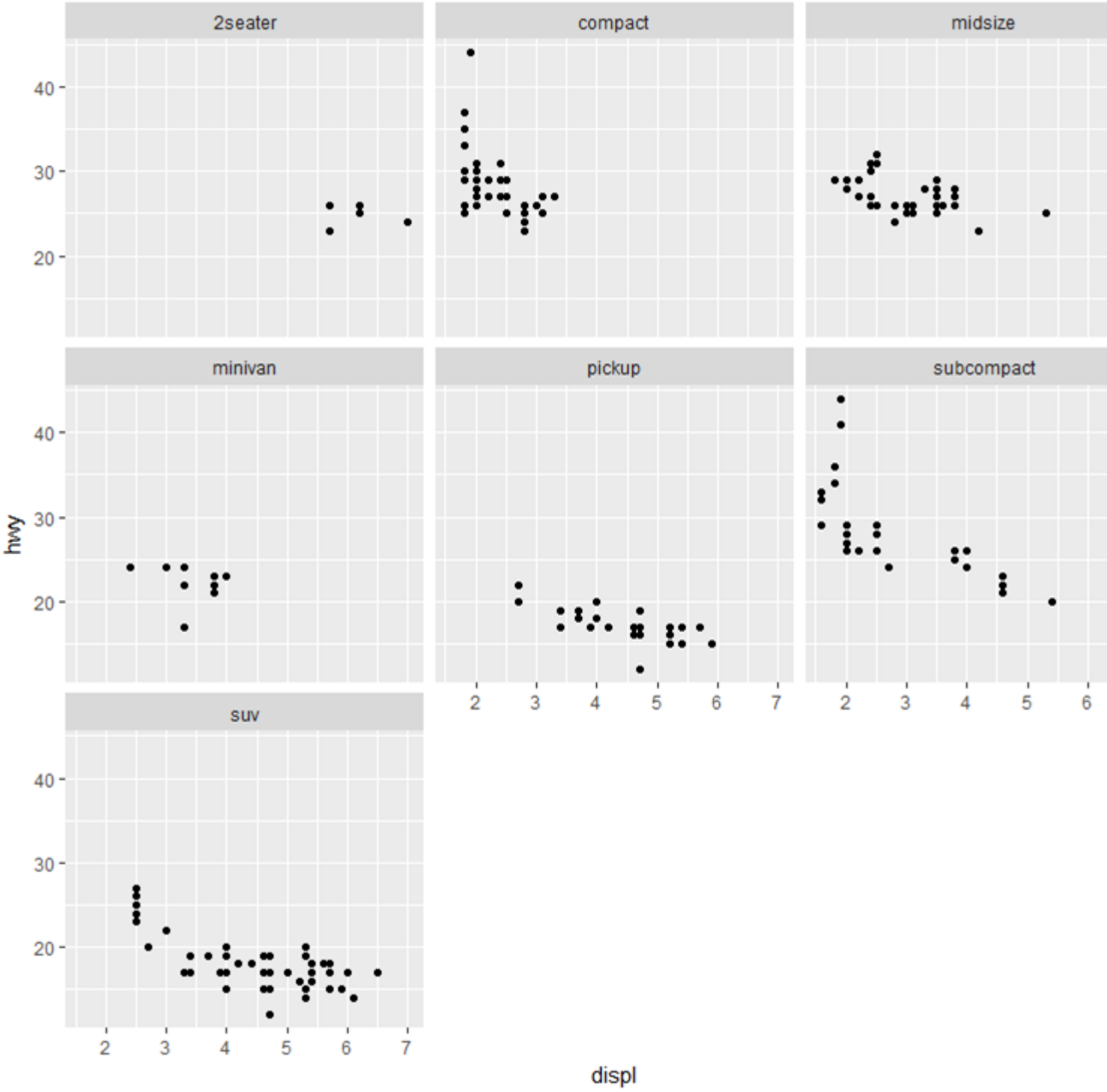
Displaying multiple plots

Display multiple plots in one image with the different `facet` functions. An advantage of this method is that all axes share the same scale across charts, making it easy to compare them at a glance. We'll use the `mpg` dataset included in `ggplot2`.

Wrap charts line by line (attempts to create a square layout):

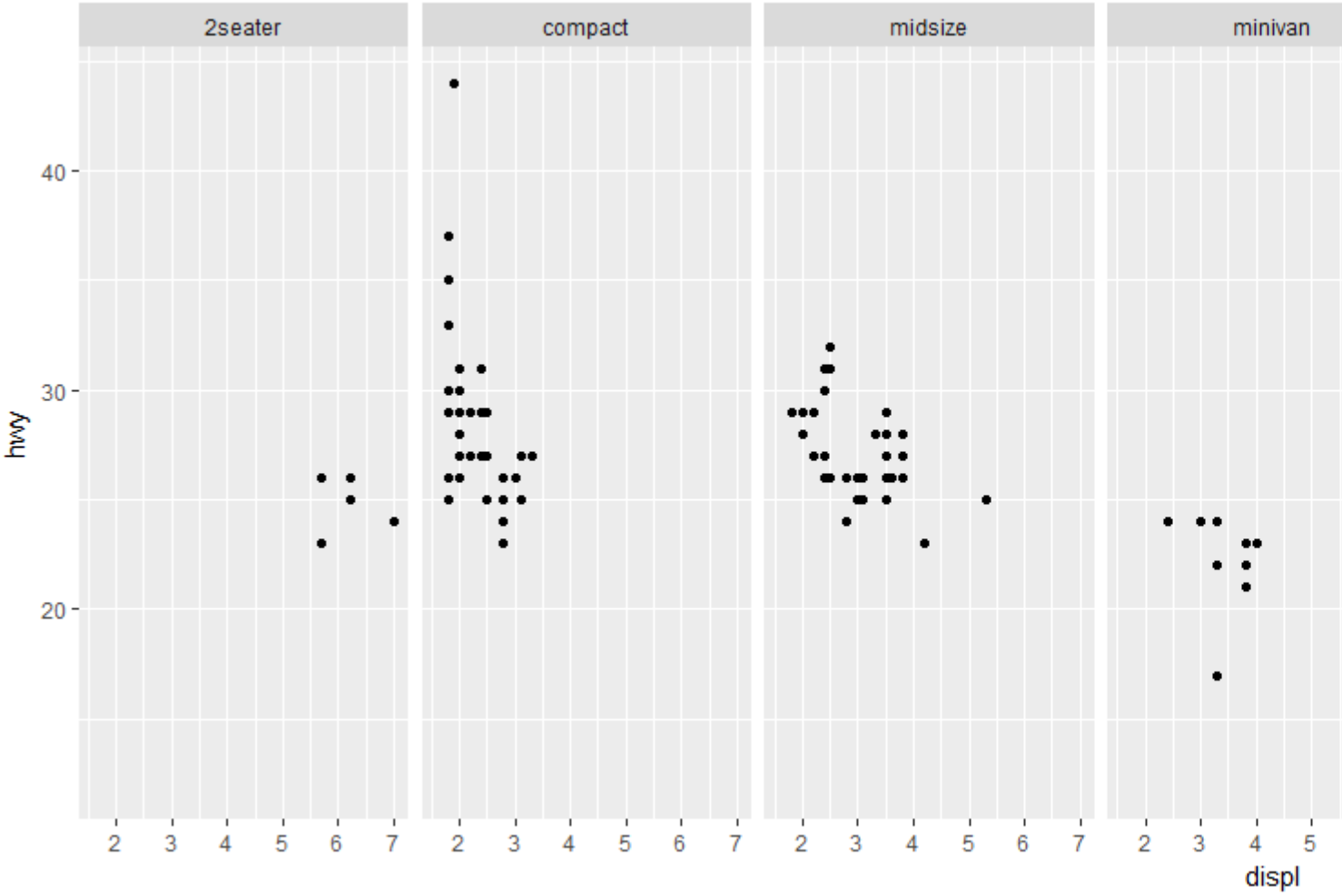
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_wrap(~class)
```


www.dbooks.org



Display multiple charts on one row, multiple columns:

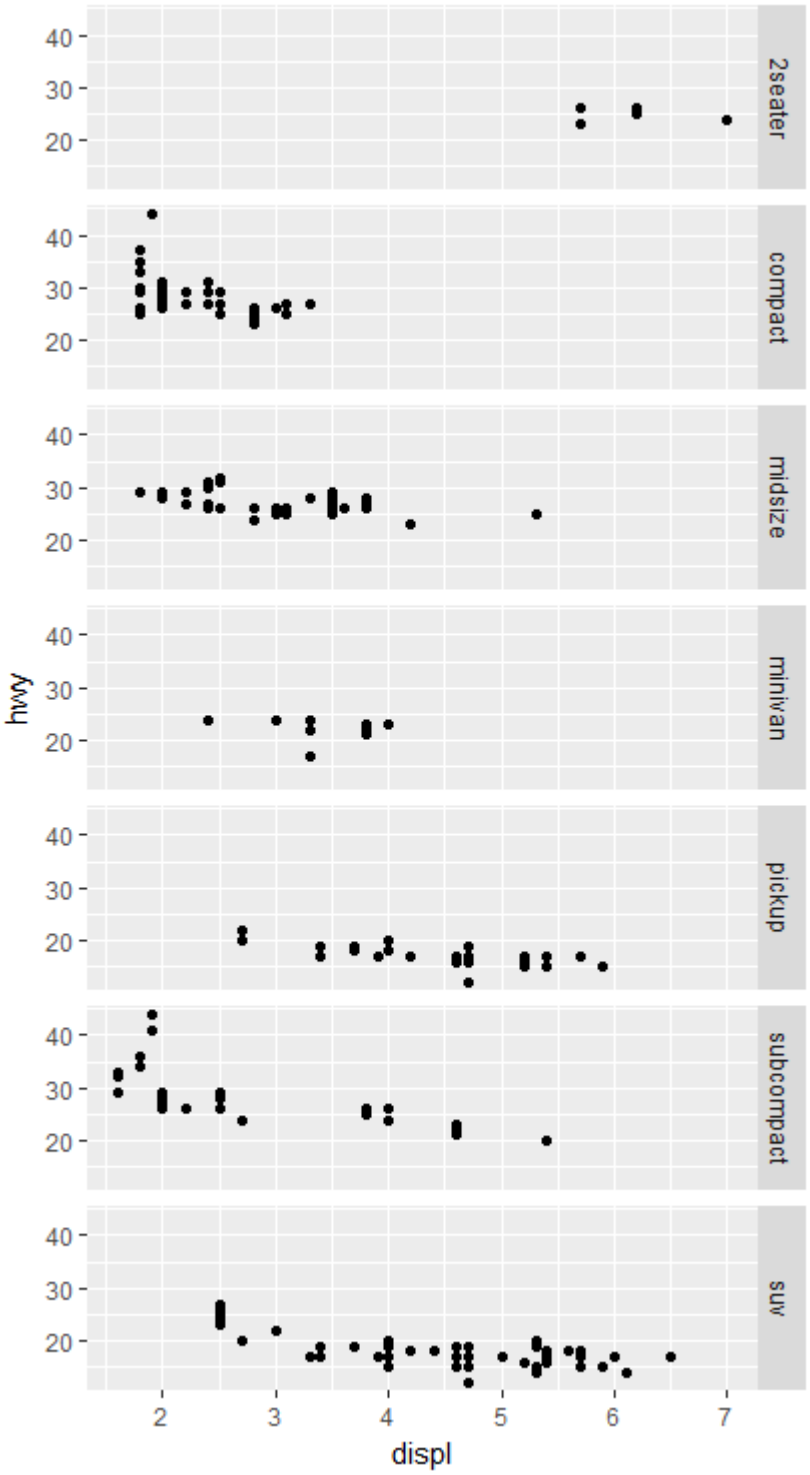
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(.~class)
```



Display multiple charts on one column, multiple rows:

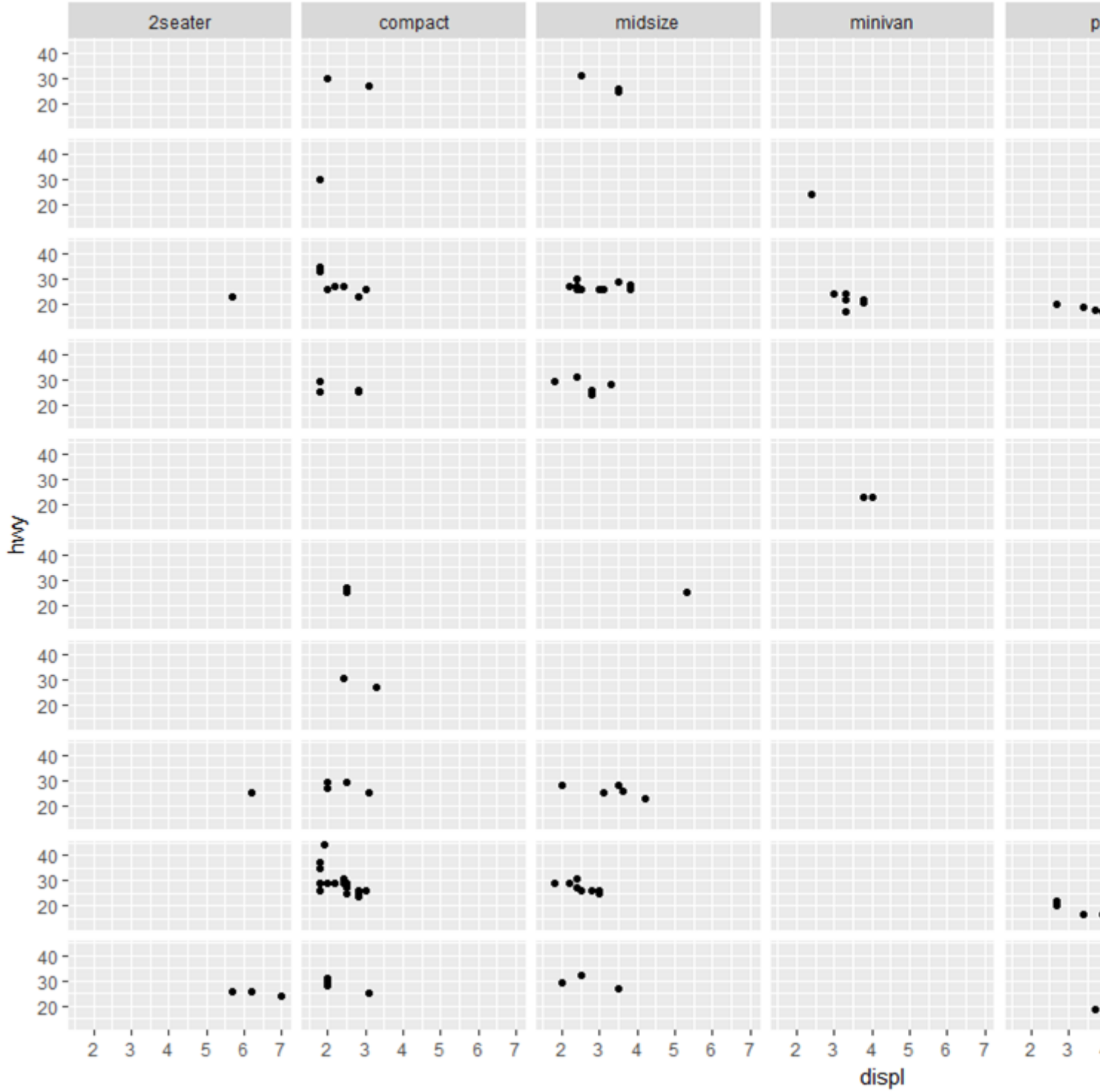
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(class~.)
```


www.dbooks.org



Display multiple charts in a grid by 2 variables:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(trans~class) #"row" parameter, then "column" parameter
```



Prepare your data for plotting

`ggplot2` works best with a long data frame. The following sample data which represents the prices for sweets on 20 different days, in a format described as wide, because each category has a column.

```
set.seed(47)
sweetsWide <- data.frame(date      = 1:20,
                         chocolate = runif(20, min = 2, max = 4),
                         iceCream  = runif(20, min = 0.5, max = 1),
                         candy     = runif(20, min = 1, max = 3))
```


www.dbooks.org

```
head(sweetsWide)
##   date chocolate  iceCream    candy
## 1    1  3.953924 0.5890727 1.117311
## 2    2  2.747832 0.7783982 1.740851
## 3    3  3.523004 0.7578975 2.196754
## 4    4  3.644983 0.5667152 2.875028
## 5    5  3.147089 0.8446417 1.733543
## 6    6  3.382825 0.6900125 1.405674
```

To convert `sweetsWide` to long format for use with `ggplot2`, several useful functions from base R, and the packages `reshape2`, `data.table` and `tidyr` (in chronological order) can be used:

```
# reshape from base R
sweetsLong <- reshape(sweetsWide, idvar = 'date', direction = 'long',
                      varying = list(2:4), new.row.names = NULL, times = names(sweetsWide)[-
1])

# melt from 'reshape2'
library(reshape2)
sweetsLong <- melt(sweetsWide, id.vars = 'date')

# melt from 'data.table'
# which is an optimized & extended version of 'melt' from 'reshape2'
library(data.table)
sweetsLong <- melt(setDT(sweetsWide), id.vars = 'date')

# gather from 'tidyr'
library(tidyr)
sweetsLong <- gather(sweetsWide, sweet, price, chocolate:candy)
```

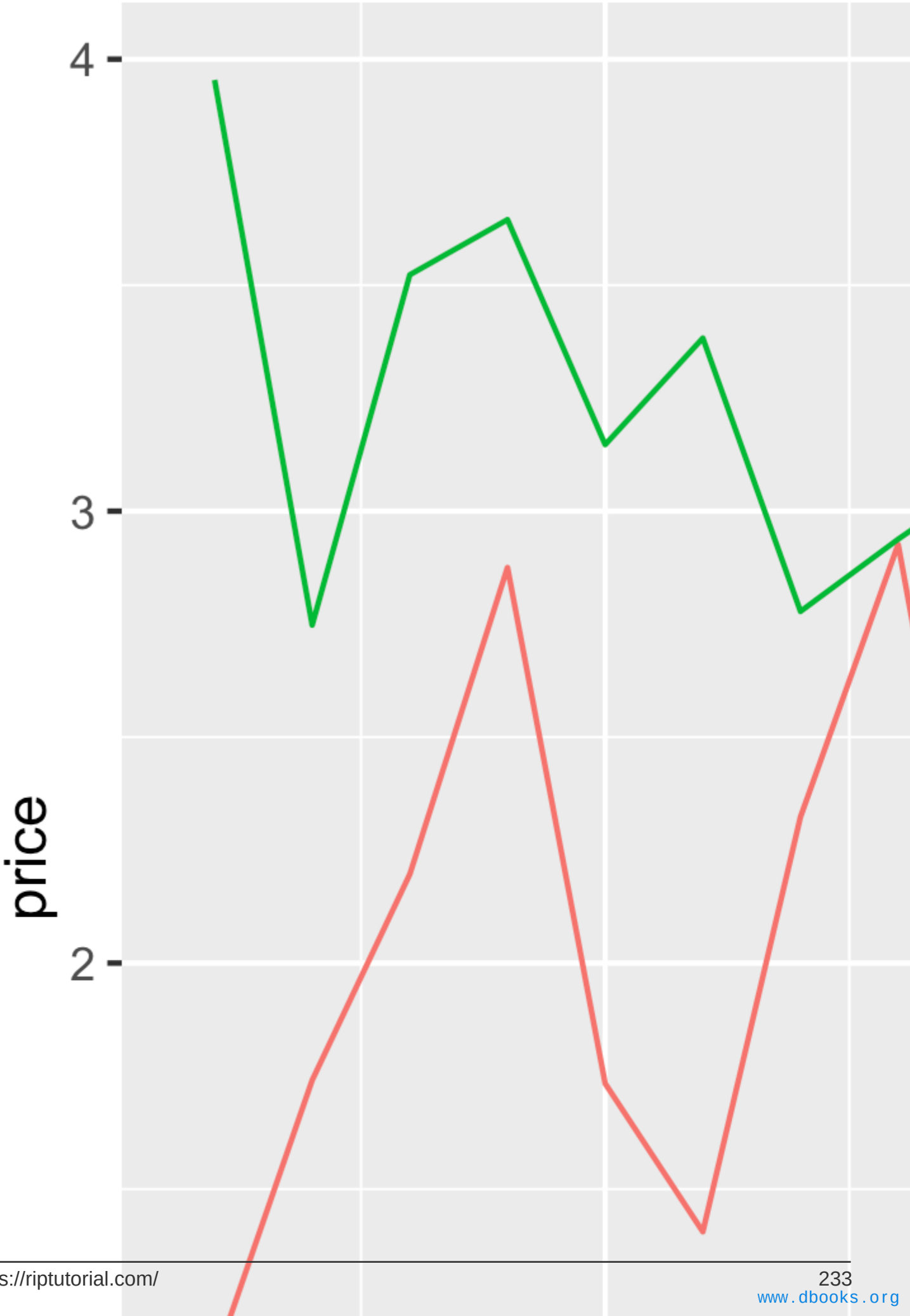
The all give a similar result:

```
head(sweetsLong)
##   date     sweet    price
## 1    1 chocolate 3.953924
## 2    2 chocolate 2.747832
## 3    3 chocolate 3.523004
## 4    4 chocolate 3.644983
## 5    5 chocolate 3.147089
## 6    6 chocolate 3.382825
```

See also Reshaping data between long and wide forms for details on converting data between *long* and *wide* format.

The resulting `sweetsLong` has one column of prices and one column describing the type of sweet. Now plotting is much simpler:

```
library(ggplot2)
ggplot(sweetsLong, aes(x = date, y = price, colour = sweet)) + geom_line()
```

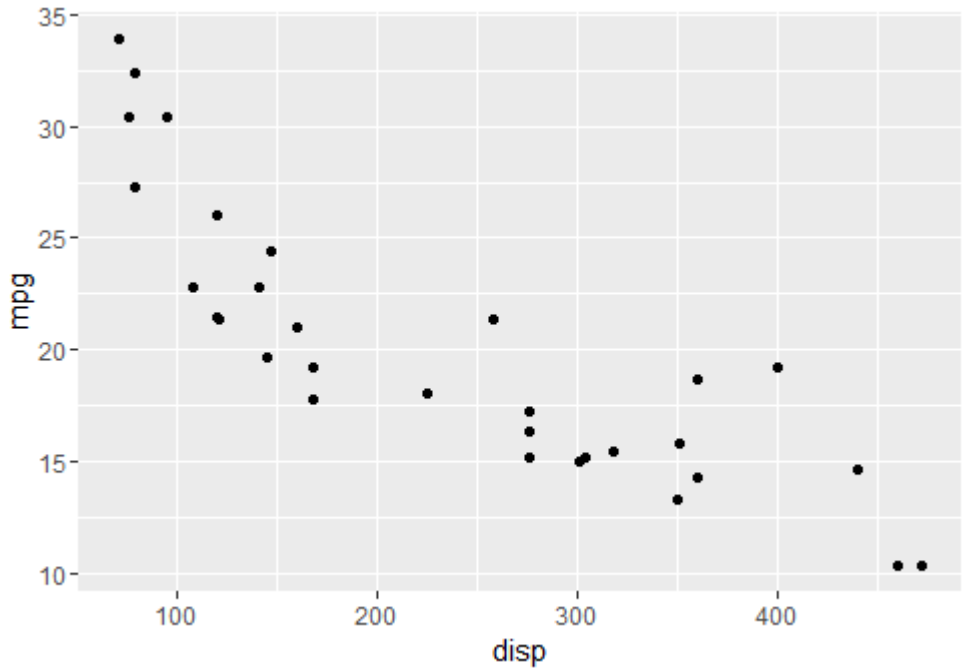


www.dbooks.org

function, trying to always plot out your data without requiring too much specifications.

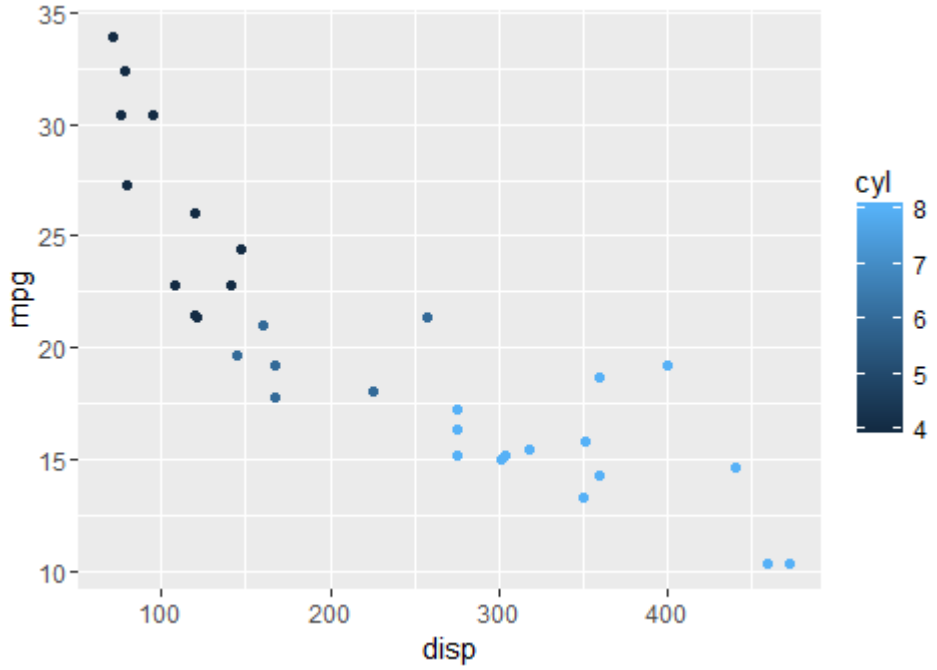
basic qplot

```
qplot(x = disp, y = mpg, data = mtcars)
```



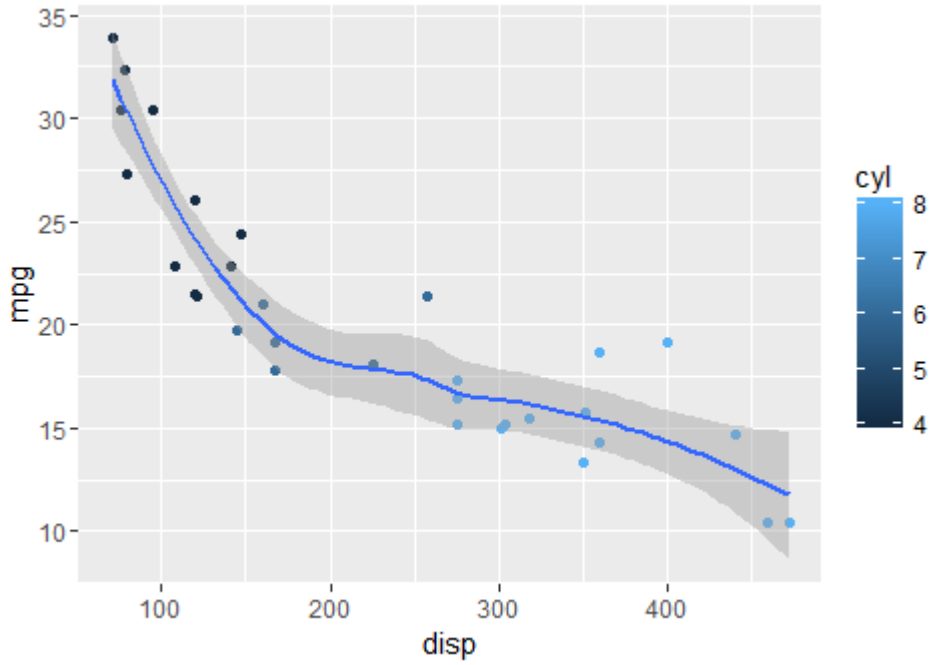
adding colors

```
qplot(x = disp, y = mpg, colour = cyl,data = mtcars)
```



adding a smoother

```
qplot(x = disp, y = mpg, geom = c("point", "smooth"), data = mtcars)
```



Read ggplot2 online: https://riptutorial.com/r/topic/1334/ggplot2

Chapter 44: GPU-accelerated computing

Remarks

GPU computing requires a 'platform' which can connect to and utilize the hardware. The two primary low-level languages that accomplish this are CUDA and OpenCL. The former requires installation of the proprietary NVIDIA CUDA Toolkit and is only applicable on NVIDIA GPUs. The latter is both company (e.g. NVIDIA, AMD, Intel) and hardware independent (CPU or GPU) but requires the installation of an SDK (software development kit). In order to use a GPU via R you will need to install one of these pieces of software first.

Once either the CUDA Toolkit or a OpenCL SDK is installed, you can install an appropriate R package. Almost all the R GPU packages are dependent upon CUDA and limited to NVIDIA GPUs. These include:

1. gputools
2. cudaBayesreg
3. HiPLARM
4. gmatrix

There are currently only two OpenCL enabled packages

1. OpenCL - interface from R to OpenCL
2. gpuR - general purpose library

Warning - installation can be difficult for different operating systems with different environmental variables and GPU platforms.

Examples

gpuR gpuMatrix objects

```
library(gpuR)

# gpuMatrix objects
X <- gpuMatrix(rnorm(100), 10, 10)
Y <- gpuMatrix(rnorm(100), 10, 10)

# transfer data to GPU when operation called
# automatically copied back to CPU
Z <- X %*% Y
```

gpuR vclMatrix objects

```
library(gpuR)
```

```
# vclMatrix objects
X <- vclMatrix(rnorm(100), 10, 10)
Y <- vclMatrix(rnorm(100), 10, 10)

# data always on GPU
# no data transfer
Z <- X %*% Y
```

Read GPU-accelerated computing online: https://riptutorial.com/r/topic/4680/gpu-accelerated-computing

www.dbooks.org

Chapter 45: Hashmaps

Examples

Environments as hash maps

*Note: in the subsequent passages, the terms **hash map** and **hash table** are used interchangeably and refer to the same concept, namely, a data structure providing efficient key lookup through use of an internal hash function.*

Introduction

Although R does not provide a native hash table structure, similar functionality can be achieved by leveraging the fact that the `environment` object returned from `new.env` (by default) provides hashed key lookups. The following two statements are equivalent, as the `hash` parameter defaults to `TRUE`:

```
H <- new.env(hash = TRUE)
H <- new.env()
```

Additionally, one may specify that the internal hash table is pre-allocated with a particular size via the `size` parameter, which has a default value of 29. Like all other R objects, `environment`s manage their own memory and will grow in capacity as needed, so while it is not necessary to request a non-default value for `size`, there may be a slight performance advantage in doing so **if** the object will (eventually) contain a very large number of elements. It is worth noting that allocating extra space via `size` does not, in itself, result in an object with a larger memory footprint:

```
object.size(new.env())
# 56 bytes

object.size(new.env(size = 10e4))
# 56 bytes
```

Insertion

Insertion of elements may be done using either of the `[[<-` or `$<-` methods provided for the `environment` class, *but **not** by using "single bracket" assignment (`[<-`)*:

```
H <- new.env()

H[["key"]] <- rnorm(1)

key2 <- "xyz"
H[[key2]] <- data.frame(x = 1:3, y = letters[1:3])

H$another_key <- matrix(rbinom(9, 1, 0.5) > 0, nrow = 3)
```

```
H["error"] <- 42
#Error in H["error"] <- 42 :
#  object of type 'environment' is not subsettable
```

Like other facets of R, the first method (`object[[key]] <- value`) is generally preferred to the second (`object$key <- value`) because in the former case, a variable maybe be used instead of a literal value (e.g `key2` in the example above).

As is generally the case with hash map implementations, the `environment` object will **not** store duplicate keys. Attempting to insert a key-value pair for an existing key will replace the previously stored value:

```
H[["key3"]] <- "original value"

H[["key3"]] <- "new value"

H[["key3"]]
#[1] "new value"
```

Key Lookup

Likewise, elements may be accessed with `[[` or `$`, but not with `[`:

```
H[["key"]]
#[1] 1.630631

H[[key2]]   ## assuming key2 <- "xyz"
#   x y
# 1 1 a
# 2 2 b
# 3 3 c

H$another_key
#       [,1]  [,2]  [,3]
# [1,]  TRUE  TRUE  TRUE
# [2,] FALSE FALSE FALSE
# [3,]  TRUE  TRUE  TRUE

H[1]
#Error in H[1] : object of type 'environment' is not subsettable
```

Inspecting the Hash Map

Being just an ordinary `environment`, the hash map can be inspected by typical means:

```
names(H)
#[1] "another_key" "xyz"         "key"         "key3"

ls(H)
#[1] "another_key" "key"         "key3"        "xyz"
```


www.dbooks.org

```
str(H)
#<environment: 0x7828228>

ls.str(H)
# another_key :  logi [1:3, 1:3] TRUE FALSE TRUE TRUE FALSE TRUE ...
# key :  num 1.63
# key3 :  chr "new value"
# xyz : 'data.frame':    3 obs. of  2 variables:
#  $ x: int  1 2 3
#  $ y: chr  "a" "b" "c"
```

Elements can be removed using `rm`:

```
rm(list = c("key", "key3"), envir = H)

ls.str(H)
# another_key :  logi [1:3, 1:3] TRUE FALSE TRUE TRUE FALSE TRUE ...
# xyz : 'data.frame':    3 obs. of  2 variables:
#  $ x: int  1 2 3
#  $ y: chr  "a" "b" "c"
```

Flexibility

One of the major benefits of using `environment` objects as hash tables is their ability to store virtually any type of object as a value, *even other* *`environment`*s:

```
H2 <- new.env()

H2[["a"]] <- LETTERS
H2[["b"]] <- as.list(x = 1:5, y = matrix(rnorm(10), 2))
H2[["c"]] <- head(mtcars, 3)
H2[["d"]] <- Sys.Date()
H2[["e"]] <- Sys.time()
H2[["f"]] <- (function() {
    H3 <- new.env()
    for (i in seq_along(names(H2))) {
        H3[[names(H2)[i]]] <- H2[[names(H2)[i]]]
    }
    H3
})()

ls.str(H2)
# a :  chr [1:26] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" ...
# b : List of 5
#  $ : int 1
#  $ : int 2
#  $ : int 3
#  $ : int 4
#  $ : int 5
# c : 'data.frame':    3 obs. of  11 variables:
#  $ mpg : num  21 21 22.8
#  $ cyl : num  6 6 4
#  $ disp: num  160 160 108
#  $ hp  : num  110 110 93
#  $ drat: num  3.9 3.9 3.85
#  $ wt  : num  2.62 2.88 2.32
```

```
#  $ qsec: num  16.5 17 18.6
#  $ vs  : num  0 0 1
#  $ am  : num  1 1 1
#  $ gear: num  4 4 4
#  $ carb: num  4 4 1
# d :  Date[1:1], format: "2016-08-03"
# e :  POSIXct[1:1], format: "2016-08-03 19:25:14"
# f : <environment: 0x91a7cb8>

ls.str(H2$f)
# a :  chr [1:26] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" ...
# b : List of 5
#  $ : int 1
#  $ : int 2
#  $ : int 3
#  $ : int 4
#  $ : int 5
# c : 'data.frame':    3 obs. of  11 variables:
#  $ mpg : num  21 21 22.8
#  $ cyl : num  6 6 4
#  $ disp: num  160 160 108
#  $ hp  : num  110 110 93
#  $ drat: num  3.9 3.9 3.85
#  $ wt  : num  2.62 2.88 2.32
#  $ qsec: num  16.5 17 18.6
#  $ vs  : num  0 0 1
#  $ am  : num  1 1 1
#  $ gear: num  4 4 4
#  $ carb: num  4 4 1
# d :  Date[1:1], format: "2016-08-03"
# e :  POSIXct[1:1], format: "2016-08-03 19:25:14"
```

Limitations

One of the major limitations of using `environment` objects as hash maps is that, unlike many aspects of R, vectorization is not supported for element lookup / insertion:

```
names(H2)
#[1] "a" "b" "c" "d" "e" "f"

H2[[c("a", "b")]]
#Error in H2[[c("a", "b")]] :
#  wrong arguments for subsetting an environment

Keys <- c("a", "b")
H2[[Keys]]
#Error in H2[[Keys]] : wrong arguments for subsetting an environment
```

Depending on the nature of the data being stored in the object, it may be possible to use `vapply` or `list2env` for assigning many elements at once:

```
E1 <- new.env()
invisible({
    vapply(letters, function(x) {
        E1[[x]] <- rnorm(1)
        logical(0)
```


www.dbooks.org

```
    }, FUN.VALUE = logical(0))
})

all.equal(sort(names(E1)), letters)
#[1] TRUE

Keys <- letters
E2 <- list2env(
    setNames(
        as.list(rnorm(26)),
        nm = Keys),
    envir = NULL,
    hash = TRUE
)

all.equal(sort(names(E2)), letters)
#[1] TRUE
```

Neither of the above are particularly concise, but may be preferable to using a `for` loop, etc. when the number of key-value pairs is large.

package:hash

The hash package offers a hash structure in R. However, it terms of timing for both inserts and reads it compares unfavorably to using environments as a hash. This documentation simply acknowledges its existence and provides sample timing code below for the above stated reasons. There is no identified case where hash is an appropriate solution in R code today.

Consider:

```
# Generic unique string generator
unique_strings <- function(n){
    string_i <- 1
    string_len <- 1
    ans <- character(n)
    chars <- c(letters,LETTERS)
    new_strings <- function(len,pfx){
    for(i in 1:length(chars)){
        if (len == 1){
        ans[string_i] <<- paste(pfx,chars[i],sep='')
        string_i <<- string_i + 1
        } else {
        new_strings(len-1,pfx=paste(pfx,chars[i],sep=''))
        }
        if (string_i > n) return ()
    }
    }
    while(string_i <= n){
    new_strings(string_len,'')
    string_len <- string_len + 1
    }
    sample(ans)
}

# Generate timings using an enviornment
timingsEnv <- plyr::adply(2^(10:15),.mar=1,.fun=function(i){
    strings <- unique_strings(i)
```

```
    ht1 <- new.env(hash=TRUE)
    lapply(strings, function(s){ ht1[[s]] <<- 0L})
    data.frame(
    size=c(i,i),
    seconds=c(
        system.time(for (j in 1:i) ht1[[strings[j]]]==0L)[3]),
    type = c('1_hashedEnv')
    )
})

timingsHash <- plyr::adply(2^(10:15),.mar=1,.fun=function(i){
    strings <- unique_strings(i)
    ht <- hash::hash()
    lapply(strings, function(s) ht[[s]] <<- 0L)
    data.frame(
    size=c(i,i),
    seconds=c(
        system.time(for (j in 1:i) ht[[strings[j]]]==0L)[3]),
    type = c('3_stringHash')
    )
})
```

package:listenv

Although `package:listenv` implements a list-like interface to environments, its performance relative to environments for hash-like purposes is poor on hash retrieval. However, if the indexes are numeric, it can be quite fast on retrieval. However, they have other advantages, e.g. compatibility with `package:future`. Covering this package for that purpose goes beyond the scope of the current topic. However, the timing code provided here can be used in conjunction with the example for package:hash for write timings.

```
timingsListEnv <- plyr::adply(2^(10:15),.mar=1,.fun=function(i){
    strings <- unique_strings(i)
    le <- listenv::listenv()
    lapply(strings, function(s) le[[s]] <<- 0L)
    data.frame(
    size=c(i,i),
    seconds=c(
        system.time(for (k in 1:i) le[[k]]==0L)[3]),
    type = c('2_numericListEnv')
    )
})
```

Read Hashmaps online: https://riptutorial.com/r/topic/5179/hashmaps

www.dbooks.org

Chapter 46: heatmap and heatmap.2

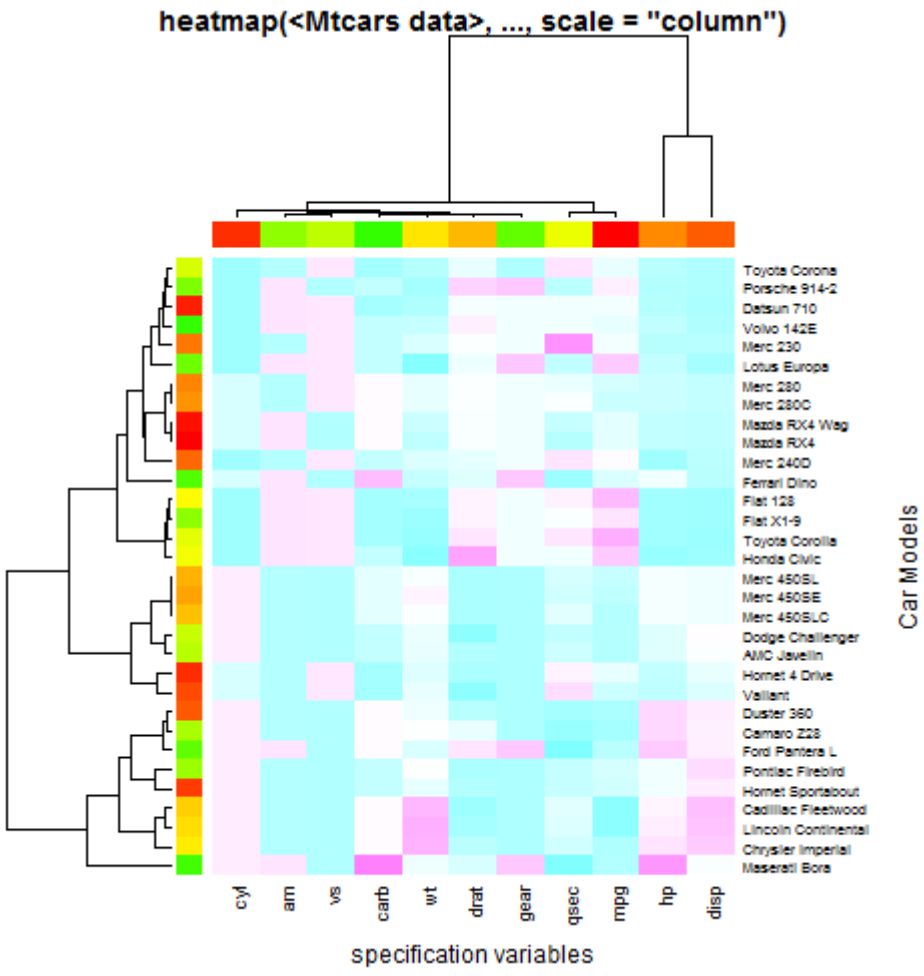
Examples

Examples from the official documentation

stats::heatmap

Example 1 (Basic usage)

```
require(graphics); require(grDevices)
x  <- as.matrix(mtcars)
rc <- rainbow(nrow(x), start = 0, end = .3)
cc <- rainbow(ncol(x), start = 0, end = .3)
hv <- heatmap(x, col = cm.colors(256), scale = "column",
              RowSideColors = rc, ColSideColors = cc, margins = c(5,10),
              xlab = "specification variables", ylab =  "Car Models",
              main = "heatmap(<Mtcars data>, ..., scale = \"column\")")
```

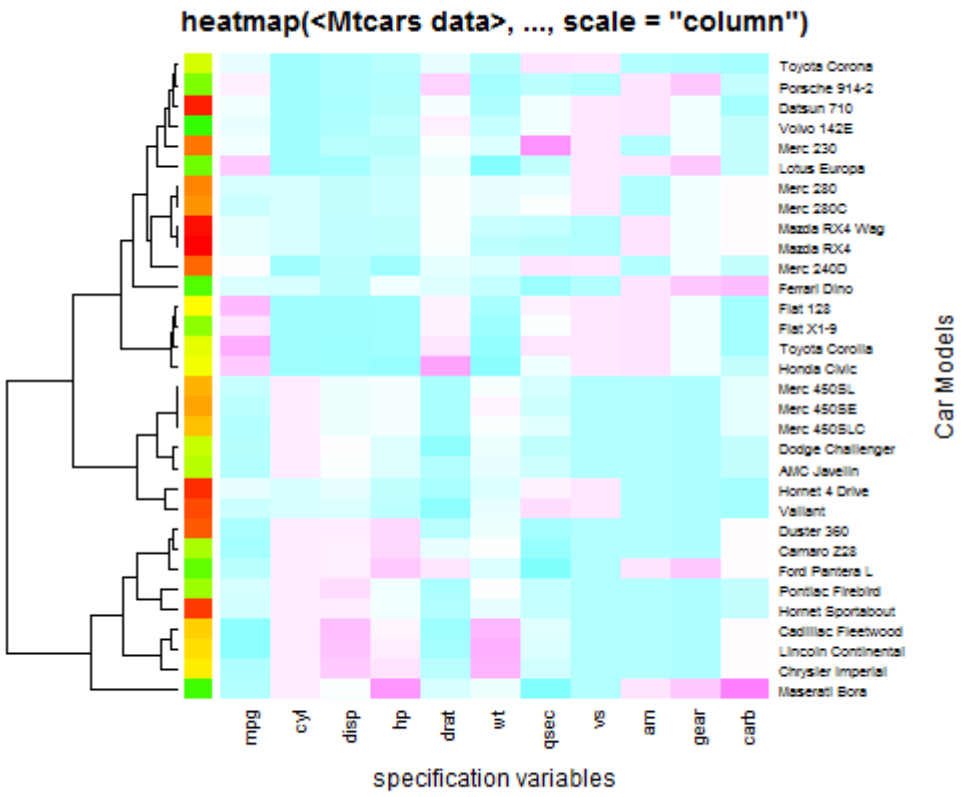



```
utils::str(hv) # the two re-ordering index vectors
# List of 4
```

```
#  $ rowInd: int [1:32] 31 17 16 15 5 25 29 24 7 6 ...
#  $ colInd: int [1:11] 2 9 8 11 6 5 10 7 1 4 ...
#  $ Rowv  : NULL
#  $ Colv  : NULL
```

Example 2 (no column dendrogram (nor reordering) at all)

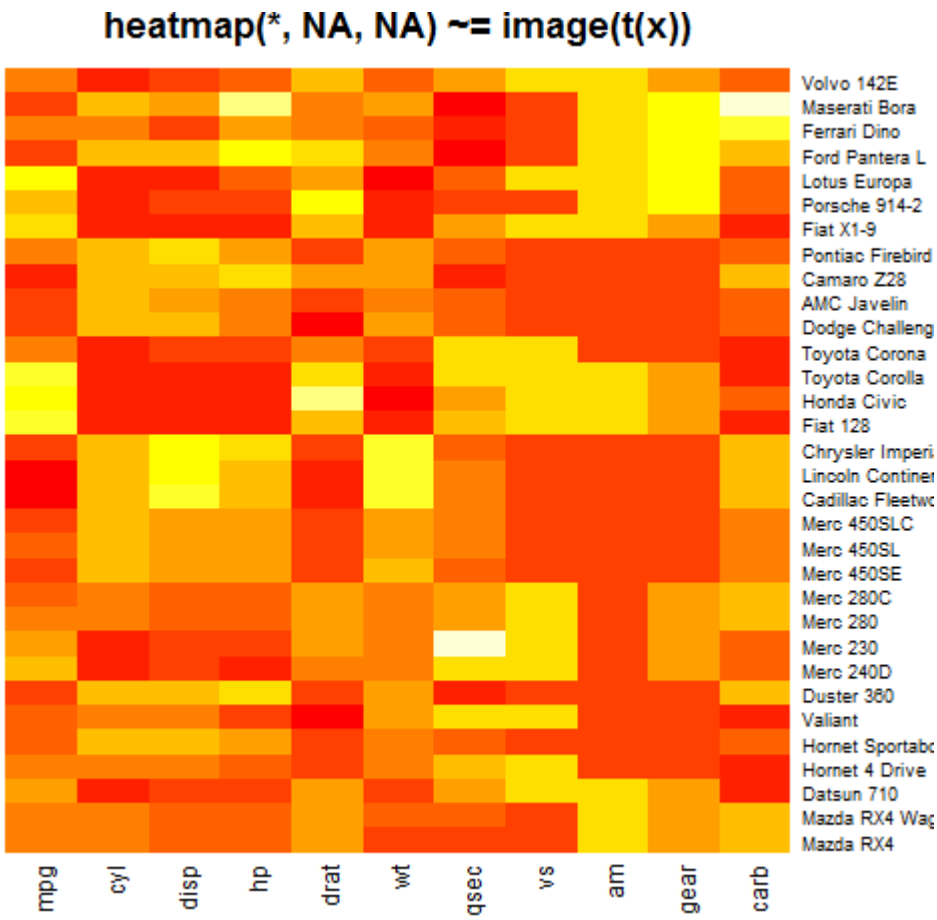
```
heatmap(x, Colv = NA, col = cm.colors(256), scale = "column",
        RowSideColors = rc, margins = c(5,10),
        xlab = "specification variables", ylab =  "Car Models",
        main = "heatmap(<Mtcars data>, ..., scale = \"column\")")
```



Example 3 ("no nothing")

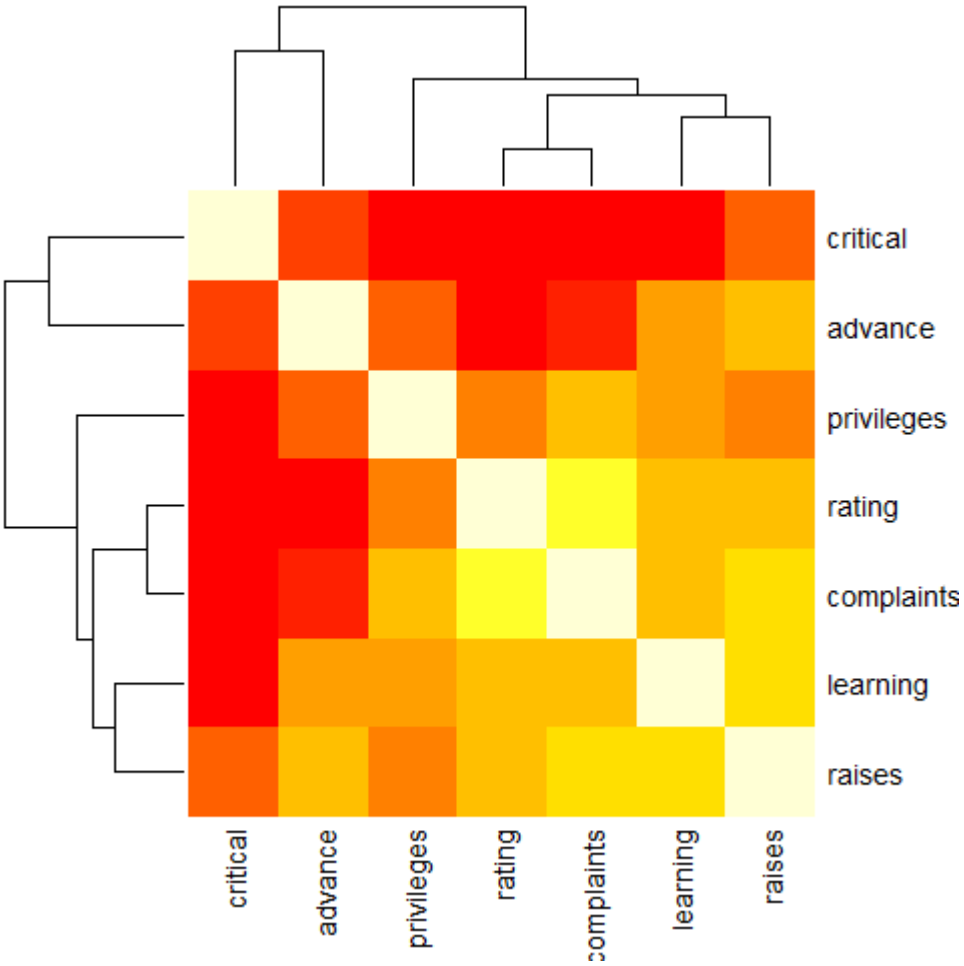
```
heatmap(x, Rowv = NA, Colv = NA, scale = "column",
        main = "heatmap(*, NA, NA) ~= image(t(x))")
```


www.dbooks.org



Example 4 (with reorder())

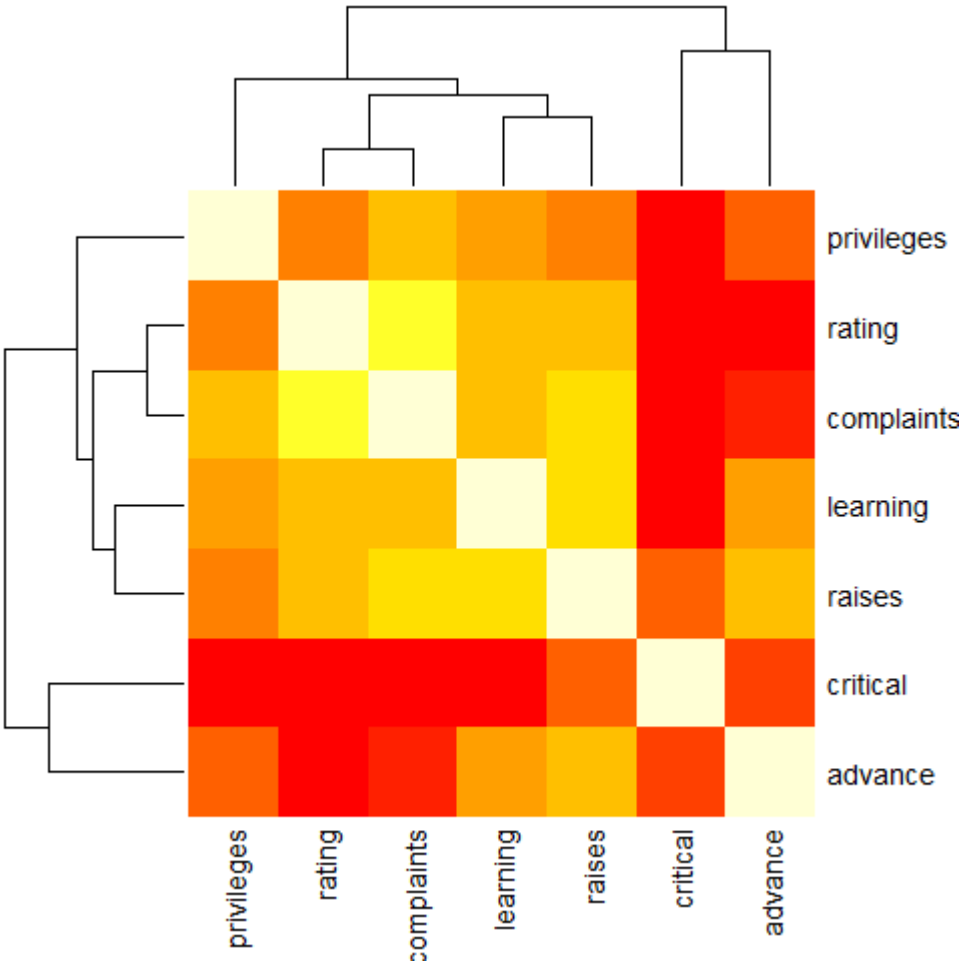
```
round(Ca <- cor(attitude), 2)
#            rating complaints privileges learning raises critical advance
# rating       1.00       0.83       0.43     0.62   0.59     0.16    0.16
# complaints   0.83       1.00       0.56     0.60   0.67     0.19    0.22
# privileges   0.43       0.56       1.00     0.49   0.45     0.15    0.34
# learning     0.62       0.60       0.49     1.00   0.64     0.12    0.53
# raises       0.59       0.67       0.45     0.64   1.00     0.38    0.57
# critical     0.16       0.19       0.15     0.12   0.38     1.00    0.28
# advance      0.16       0.22       0.34     0.53   0.57     0.28    1.00
symnum(Ca) # simple graphic
#            rt cm p l rs cr a
# rating     1
# complaints +  1
# privileges .  .  1
# learning   ,  .  . 1
# raises     .  ,  . , 1
# critical             .  1
# advance          . . .     1
# attr(,"legend")
# [1] 0 ' ' 0.3 '.' 0.6 ',' 0.8 '+' 0.9 '*' 0.95 'B' 1
heatmap(Ca,               symm = TRUE, margins = c(6,6))
```



Example 5 (*NO* reorder())

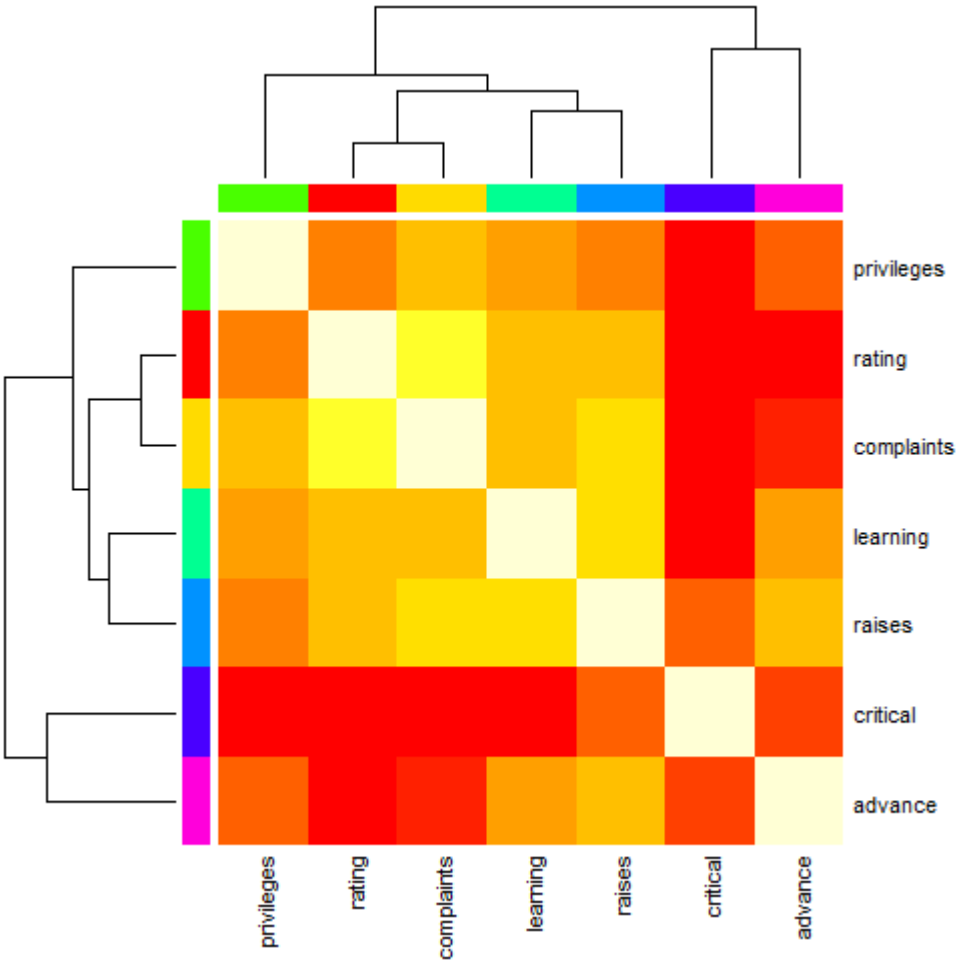
```
heatmap(Ca, Rowv = FALSE, symm = TRUE, margins = c(6,6))
```


www.dbooks.org



Example 6 (slightly artificial with color bar, without ordering)

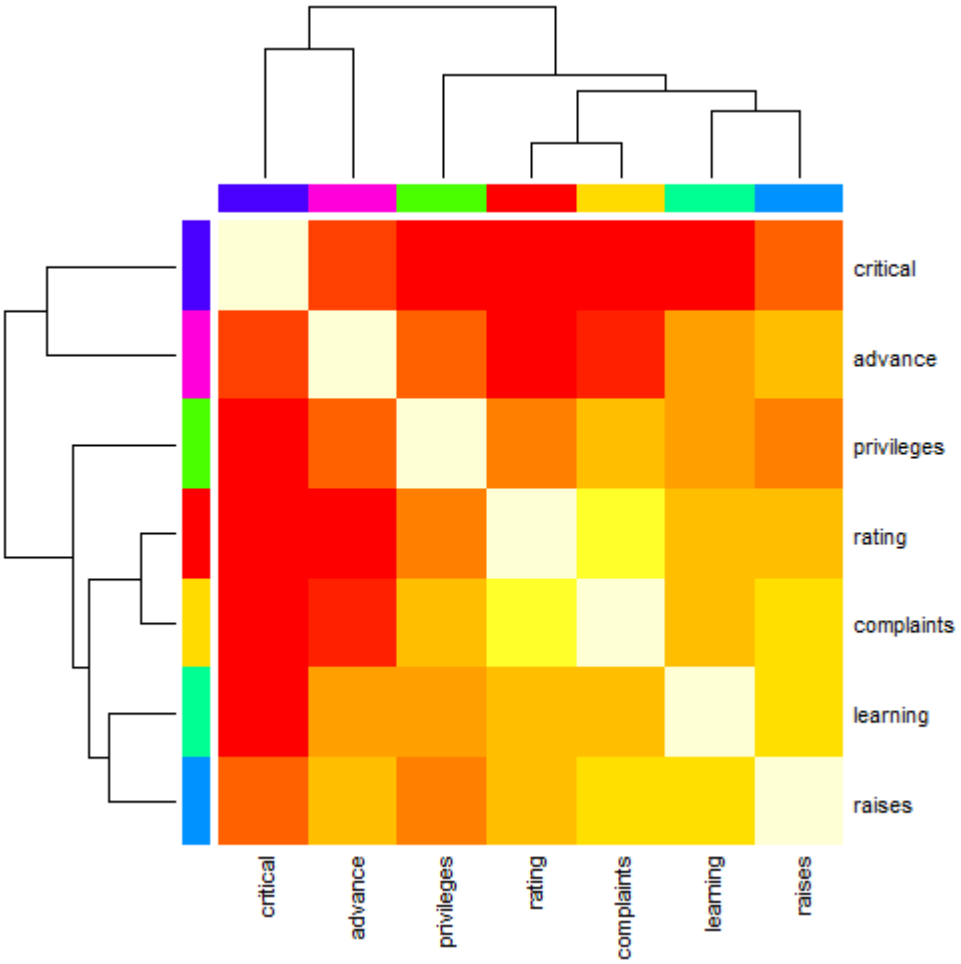
```
cc <- rainbow(nrow(Ca))
heatmap(Ca, Rowv = FALSE, symm = TRUE, RowSideColors = cc, ColSideColors = cc,
    margins = c(6,6))
```



Example 7 (slightly artificial with color bar, with ordering)

```
heatmap(Ca,        symm = TRUE, RowSideColors = cc, ColSideColors = cc,
    margins = c(6,6))
```

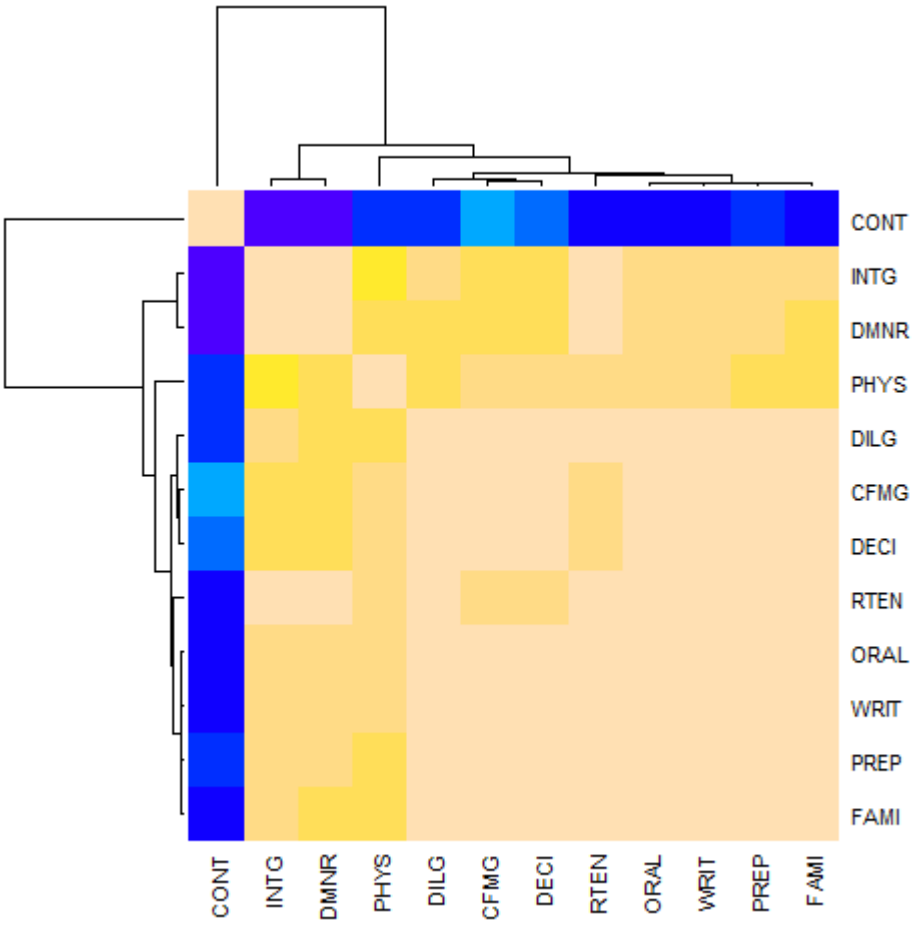

www.dbooks.org



Example 8 (For variable clustering, rather use distance based on cor())

```
symnum( cU <- cor(USJudgeRatings) )
#      CO I DM DI CF DE PR F O W PH R
# CONT 1
# INTG    1
# DMNR    B 1
# DILG    + +  1
# CFMG    + +  B  1
# DECI    + +  B  B  1
# PREP    + +  B  B  B  1
# FAMI    + +  B  *  *  B  1
# ORAL    * *  B  B  *  B  B 1
# WRIT    * +  B  *  *  B  B B 1
# PHYS    , ,  +  +  +  +  + + + 1
# RTEN    * *  *  *  *  B  * B B *  1
# attr(,"legend")
# [1] 0 ' ' 0.3 '.' 0.6 ',' 0.8 '+' 0.9 '*' 0.95 'B' 1

hU <- heatmap(cU, Rowv = FALSE, symm = TRUE, col = topo.colors(16),
              distfun = function(c) as.dist(1 - c), keep.dendro = TRUE)
```




```
## The Correlation matrix with same reordering:
round(100 * cU[hU[[1]], hU[[2]]])
#      CONT INTG DMNR PHYS DILG CFMG DECI RTEN ORAL WRIT PREP FAMI
# CONT  100  -13  -15    5    1   14    9   -3   -1   -4    1   -3
# INTG  -13  100   96   74   87   81   80   94   91   91   88   87
# DMNR  -15   96  100   79   84   81   80   94   91   89   86   84
# PHYS    5   74   79  100   81   88   87   91   89   86   85   84
# DILG    1   87   84   81  100   96   96   93   95   96   98   96
# CFMG   14   81   81   88   96  100   98   93   95   94   96   94
# DECI    9   80   80   87   96   98  100   92   95   95   96   94
# RTEN   -3   94   94   91   93   93   92  100   98   97   95   94
# ORAL   -1   91   91   89   95   95   95   98  100   99   98   98
# WRIT   -4   91   89   86   96   94   95   97   99  100   99   99
# PREP    1   88   86   85   98   96   96   95   98   99  100   99
# FAMI   -3   87   84   84   96   94   94   94   98   99   99  100
```

```
## The column dendrogram:
utils::str(hU$Colv)
# --[dendrogram w/ 2 branches and 12 members at h = 1.15]
#   |--leaf "CONT"
#   `--[dendrogram w/ 2 branches and 11 members at h = 0.258]
#      |--[dendrogram w/ 2 branches and 2 members at h = 0.0354]
#      |  |--leaf "INTG"
#      |  `--leaf "DMNR"
#      `--[dendrogram w/ 2 branches and 9 members at h = 0.187]
#         |--leaf "PHYS"
#         `--[dendrogram w/ 2 branches and 8 members at h = 0.075]
#            |--[dendrogram w/ 2 branches and 3 members at h = 0.0438]
#            |  |--leaf "DILG"
```


www.dbooks.org

```
#             |  `--[dendrogram w/ 2 branches and 2 members at h = 0.0189]
#             |     |--leaf "CFMG"
#             |     `--leaf "DECI"
#             `--[dendrogram w/ 2 branches and 5 members at h = 0.0584]
#                |--leaf "RTEN"
#                `--[dendrogram w/ 2 branches and 4 members at h = 0.0187]
#                   |--[dendrogram w/ 2 branches and 2 members at h = 0.00657]
#                   |  |--leaf "ORAL"
#                   |  `--leaf "WRIT"
#                   `--[dendrogram w/ 2 branches and 2 members at h = 0.0101]
#                      |--leaf "PREP"
#                      `--leaf "FAMI"
```

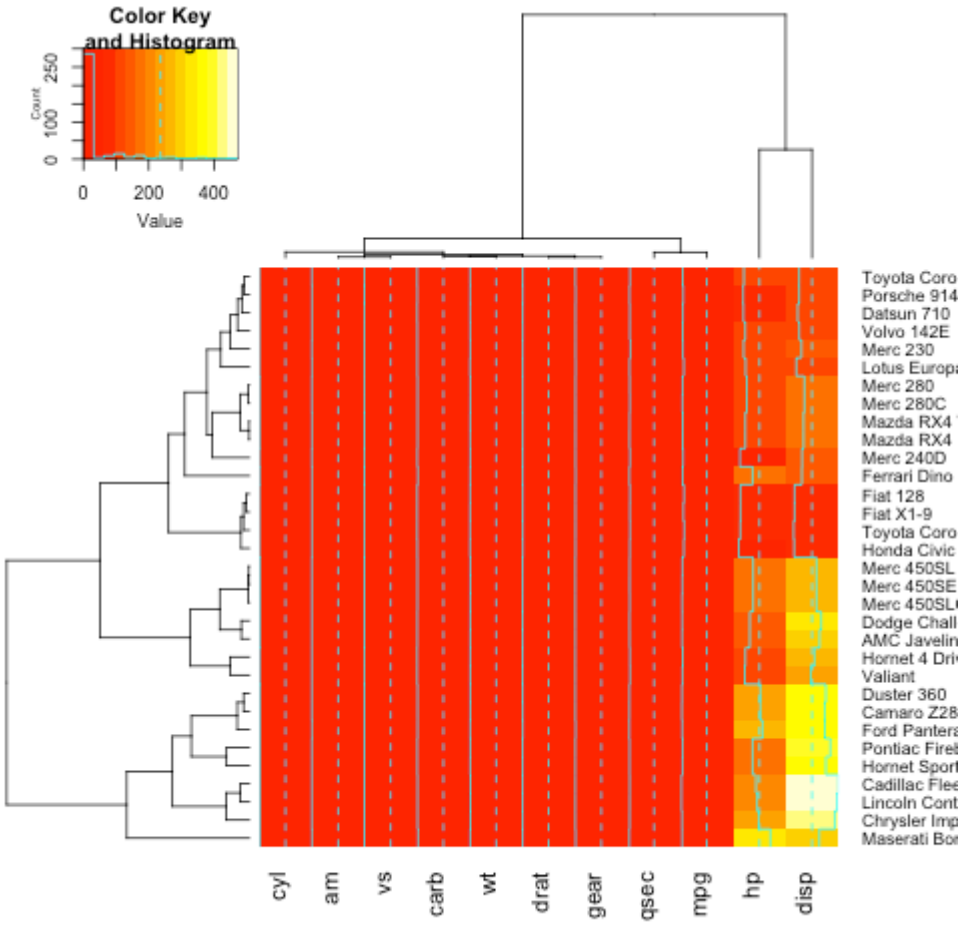
Tuning parameters in heatmap.2

Given:

```
x  <- as.matrix(mtcars)
```

One can use `heatmap.2` - a more recent optimized version of `heatmap`, by loading the following library:

```
require(gplots)
heatmap.2(x)
```

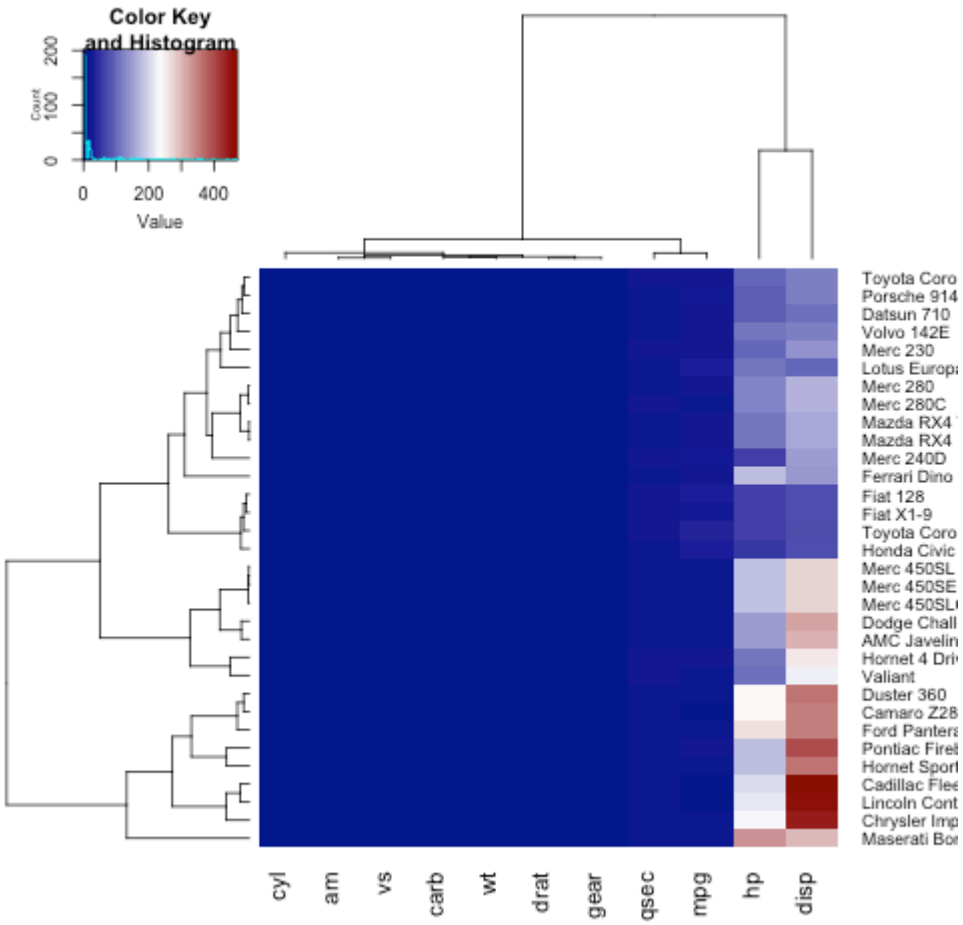


To add a title, x- or y-label to your heatmap, you need to set the `main`, `xlab` and `ylab`:

```
heatmap.2(x, main = "My main title: Overview of car features", xlab="Car features", ylab =
"Car brands")
```

If you wish to define your own color palette for your heatmap, you can set the `col` parameter by using the `colorRampPalette` function:

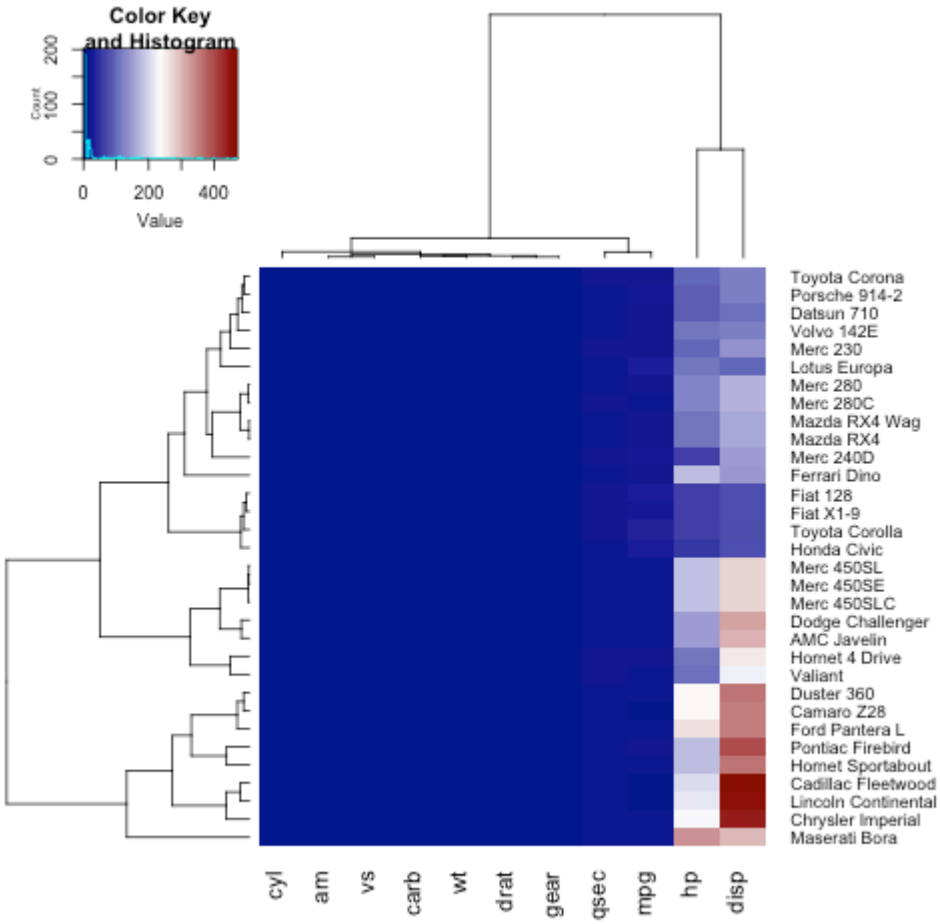
```
heatmap.2(x, trace="none", key=TRUE, Colv=FALSE,dendrogram = "row",col =
colorRampPalette(c("darkblue","white","darkred"))(100))
```



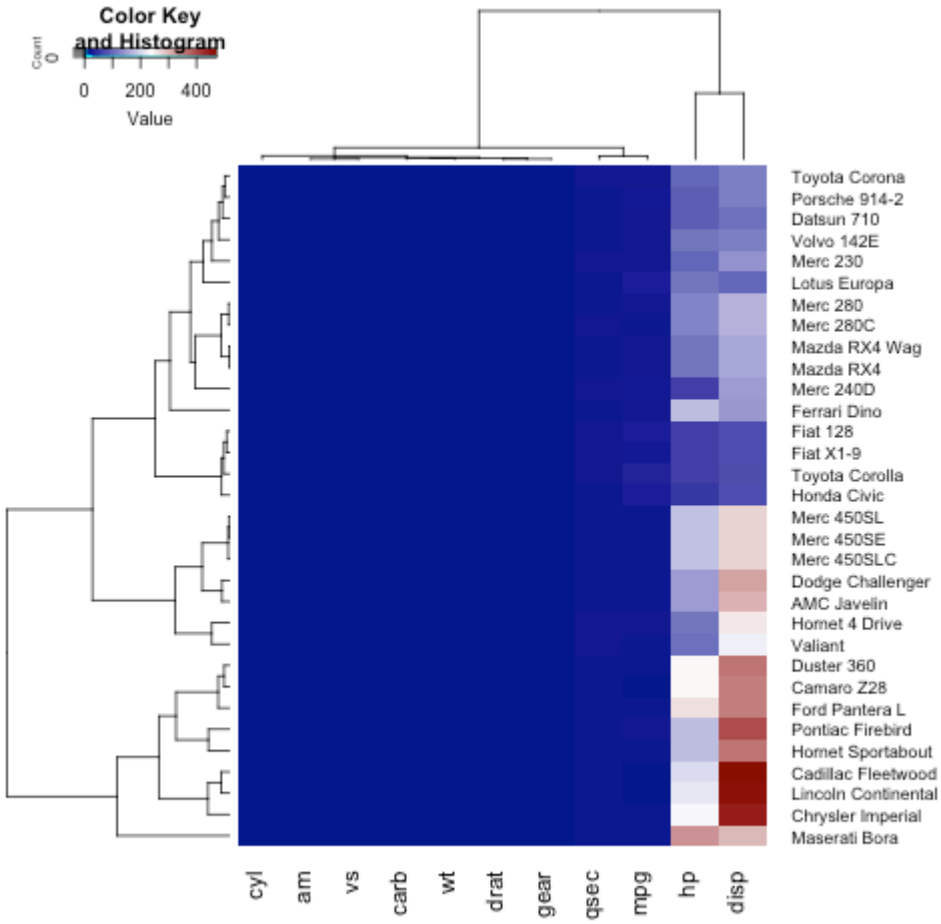
As you can notice, the labels on the y axis (the car names) don't fit in the figure. In order to fix this, the user can tune the `margins` parameter:

```
heatmap.2(x, trace="none", key=TRUE,col =
colorRampPalette(c("darkblue","white","darkred"))(100), margins=c(5,8))
```


www.dbooks.org



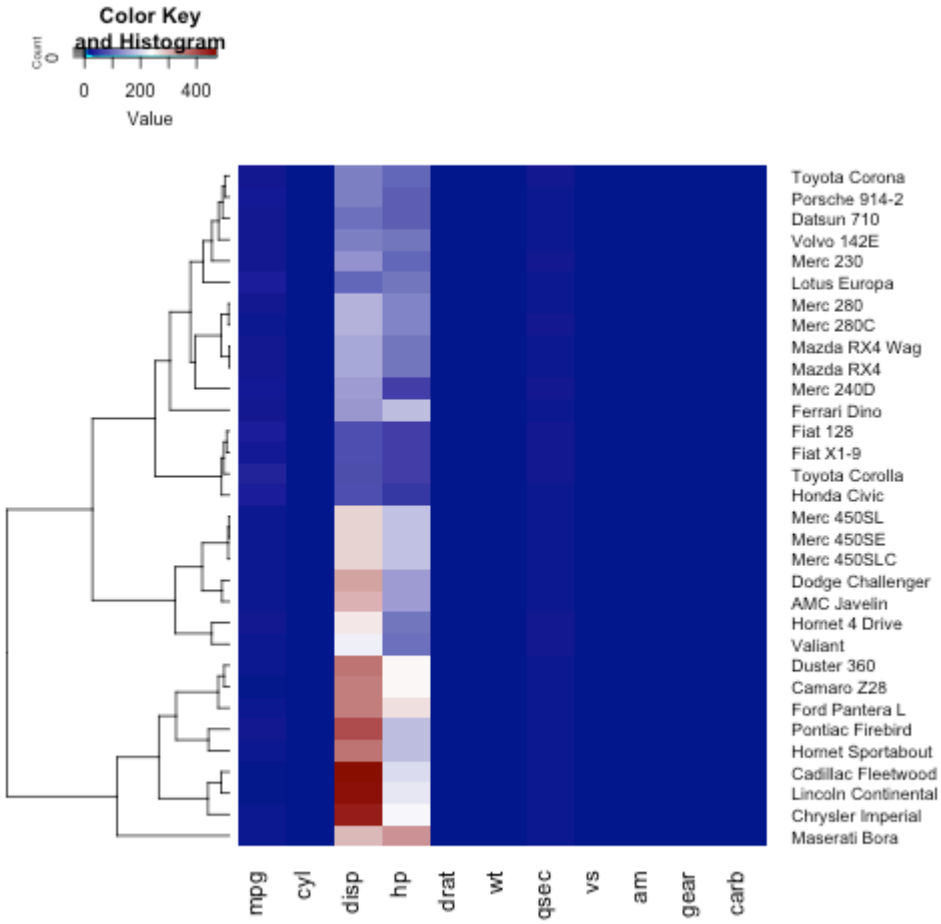
Further, we can change the dimensions of each section of our heatmap (the key histogram, the dendograms and the heatmap itself), by tuning `lhei` and `lwid` :



If we only want to show a row(or column) dendogram, we need to set `Colv=FALSE` (or `Rowv=FALSE`) and adjust the `dendogram` parameter:

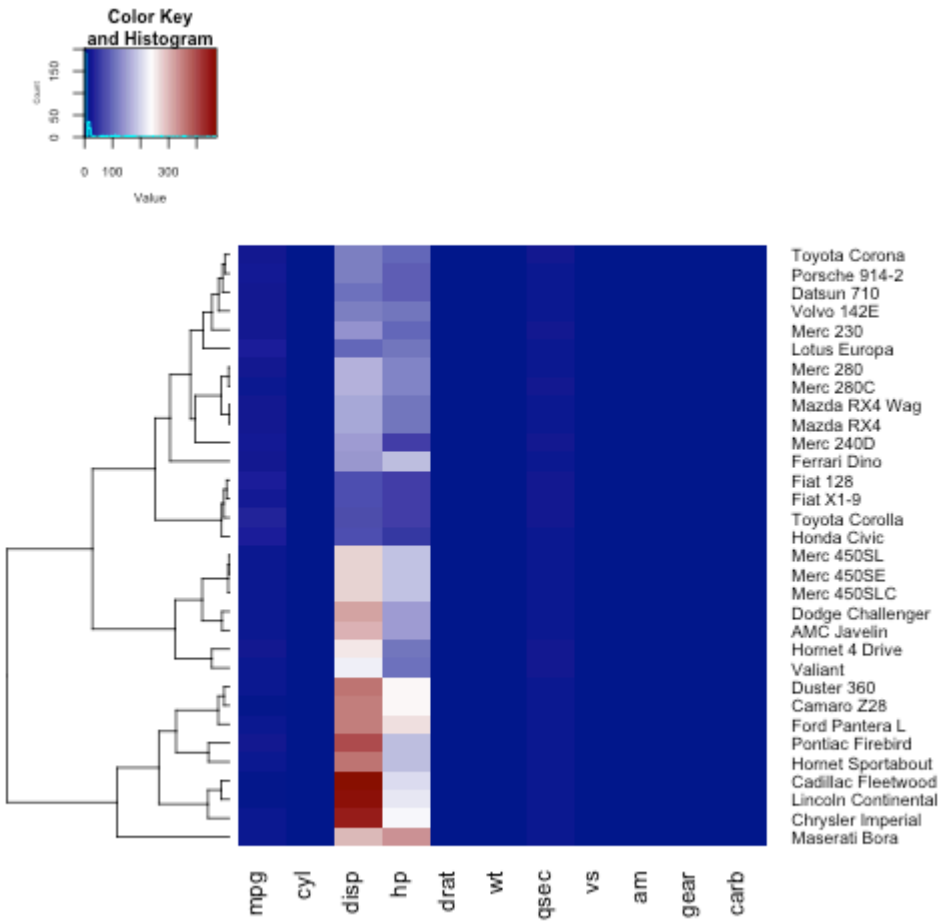
```
heatmap.2(x, trace="none", key=TRUE, Colv=FALSE, dendrogram = "row", col =
colorRampPalette(c("darkblue","white","darkred"))(100), margins=c(5,8), lwid = c(5,15), lhei =
c(3,15))
```


www.dbooks.org



For changing the font size of the legend title,labels and axis, the user needs to set `cex.main, cex.lab, cex.axis` in the `par` list:

```
par(cex.main=1, cex.lab=0.7, cex.axis=0.7)
heatmap.2(x, trace="none", key=TRUE, Colv=FALSE, dendrogram = "row", col =
colorRampPalette(c("darkblue","white","darkred"))(100), margins=c(5,8), lwid = c(5,15), lhei =
c(5,15))
```



Read heatmap and heatmap.2 online: https://riptutorial.com/r/topic/4814/heatmap-and-heatmap-2

www.dbooks.org

Chapter 47: Hierarchical clustering with hclust

Introduction

The `stats` package provides the `hclust` function to perform hierarchical clustering.

Remarks

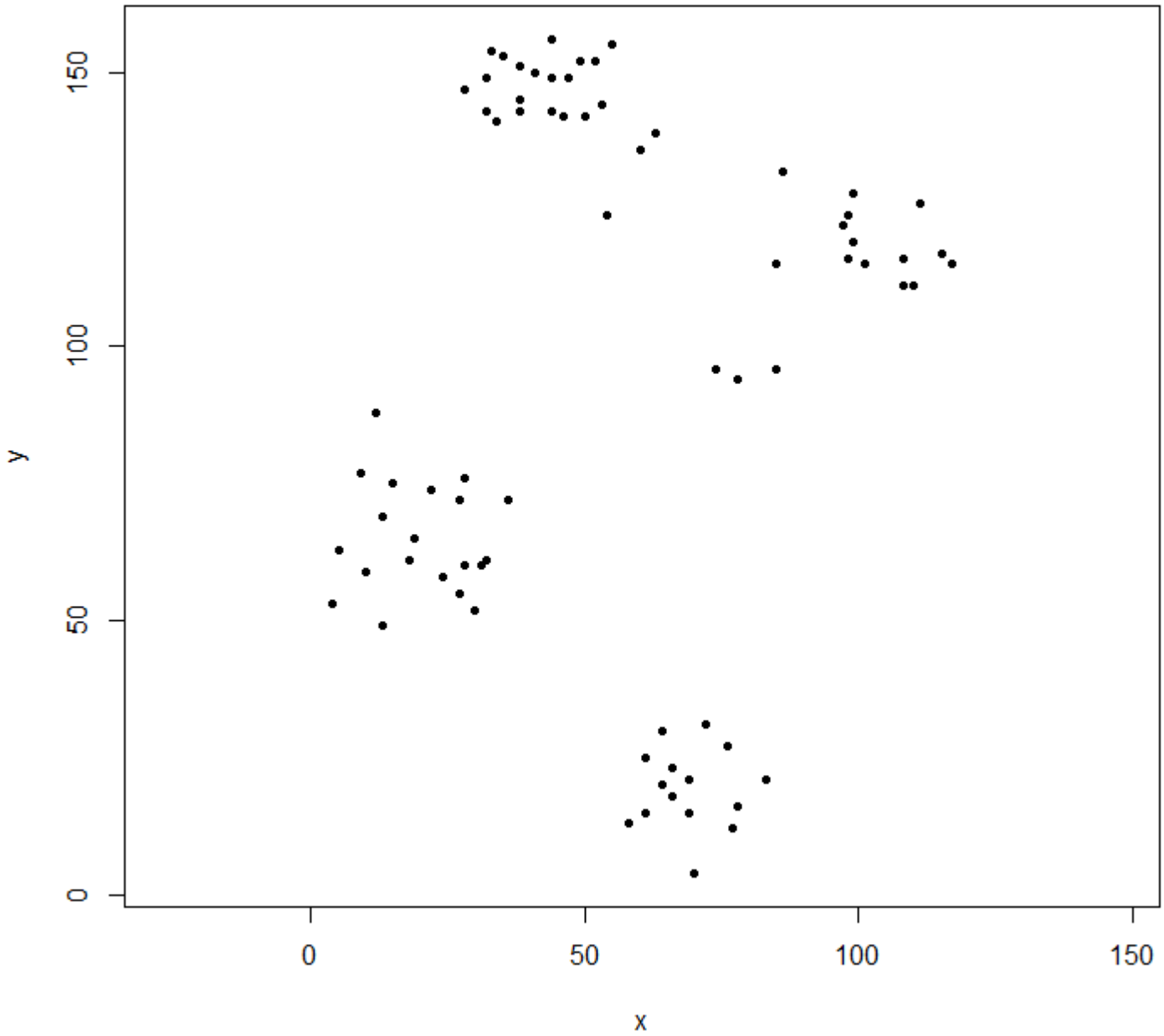
Besides hclust, other methods are available, see the CRAN Package View on Clustering.

Examples

Example 1 - Basic use of hclust, display of dendrogram, plot clusters

The cluster library contains the ruspini data - a standard set of data for illustrating cluster analysis.

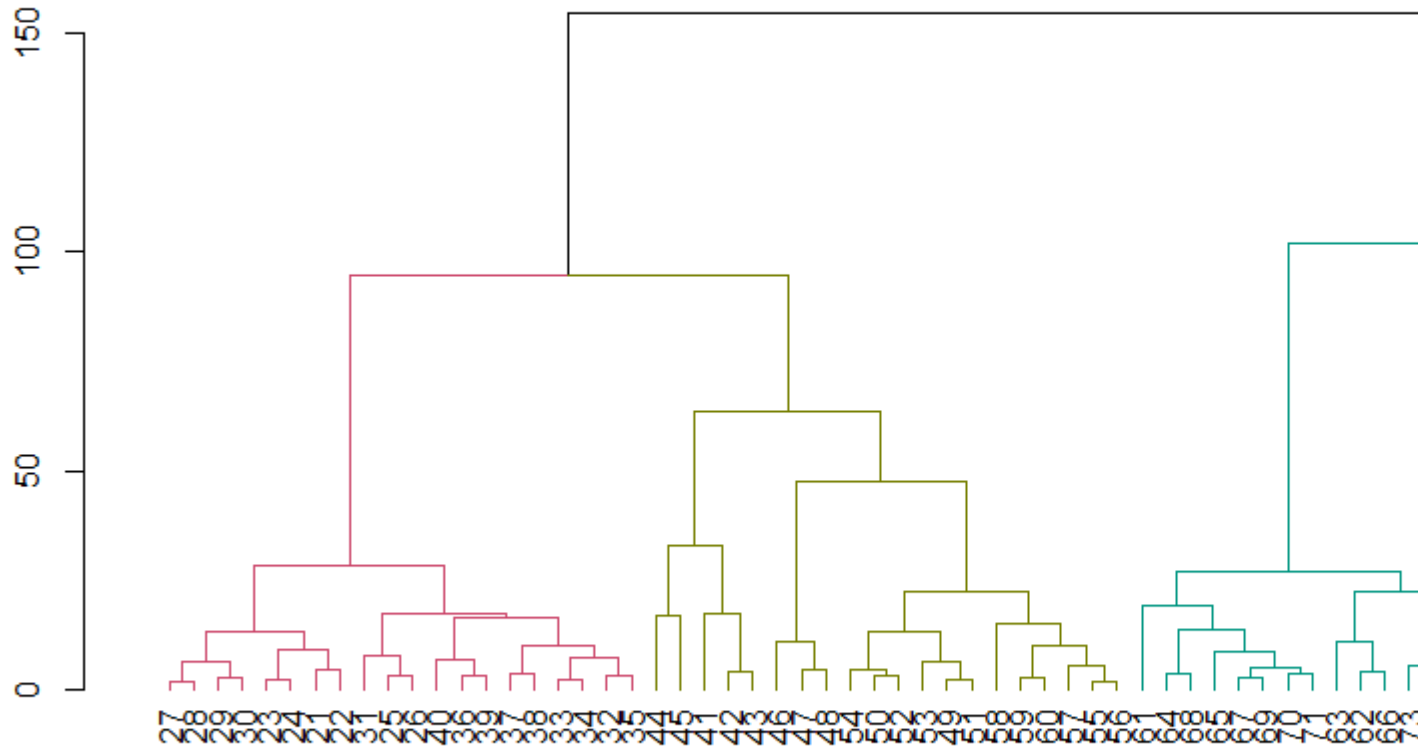
```
    library(cluster)                ## to get the ruspini data
    plot(ruspini, asp=1, pch=20)    ## take a look at the data
```



hclust expects a distance matrix, not the original data. We compute the tree using the default parameters and display it. The hang parameter lines up all of the leaves of the tree along the baseline.

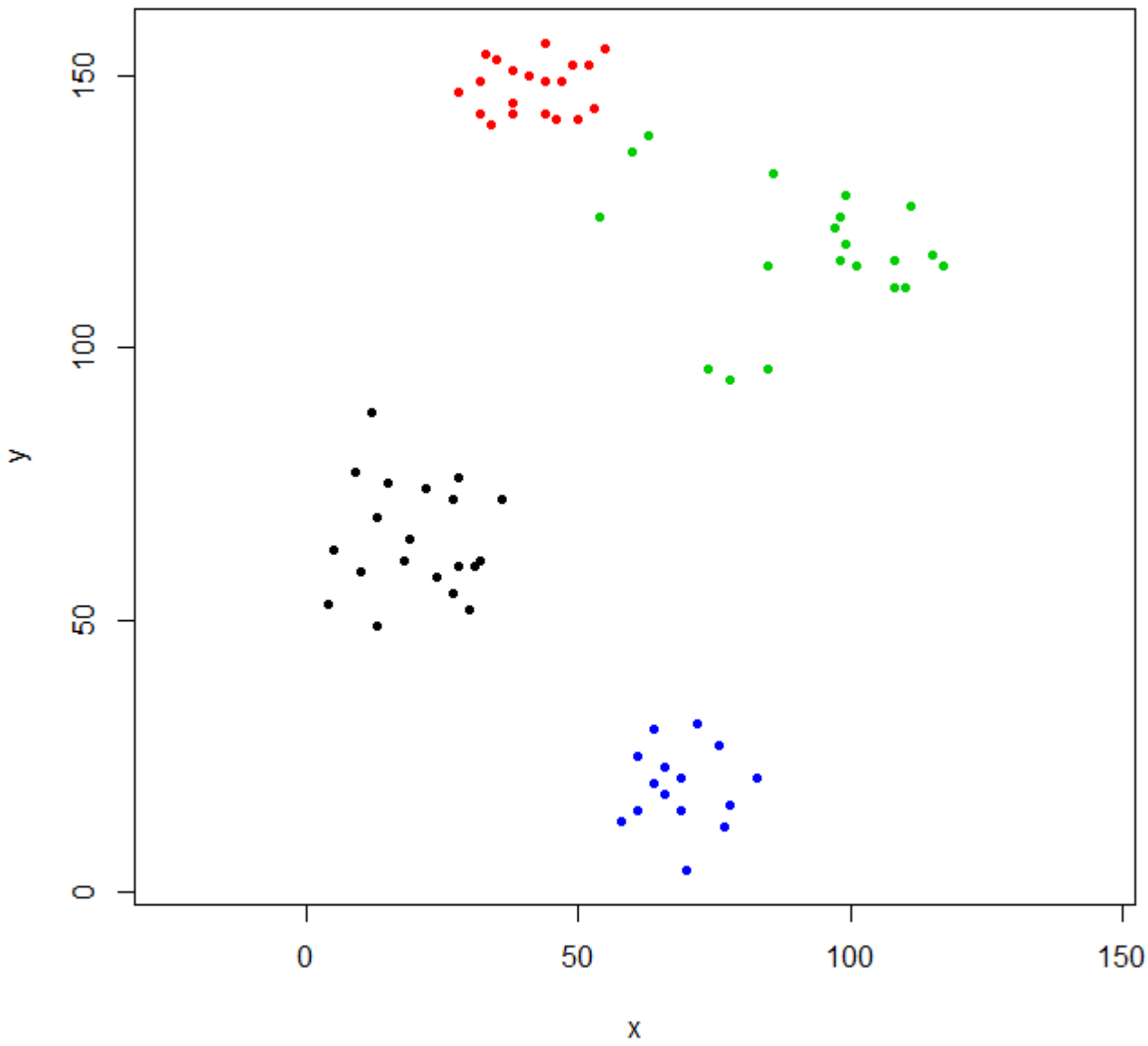
```
ruspini_hc_defaults <- hclust(dist(ruspini))
dend <- as.dendrogram(ruspini_hc_defaults)
if(!require(dendextend)) install.packages("dendextend"); library(dendextend)
dend <- color_branches(dend, k = 4)
plot(dend)
```


www.dbooks.org



Cut the tree to give four clusters and replot the data coloring the points by cluster. k is the desired number of clusters.

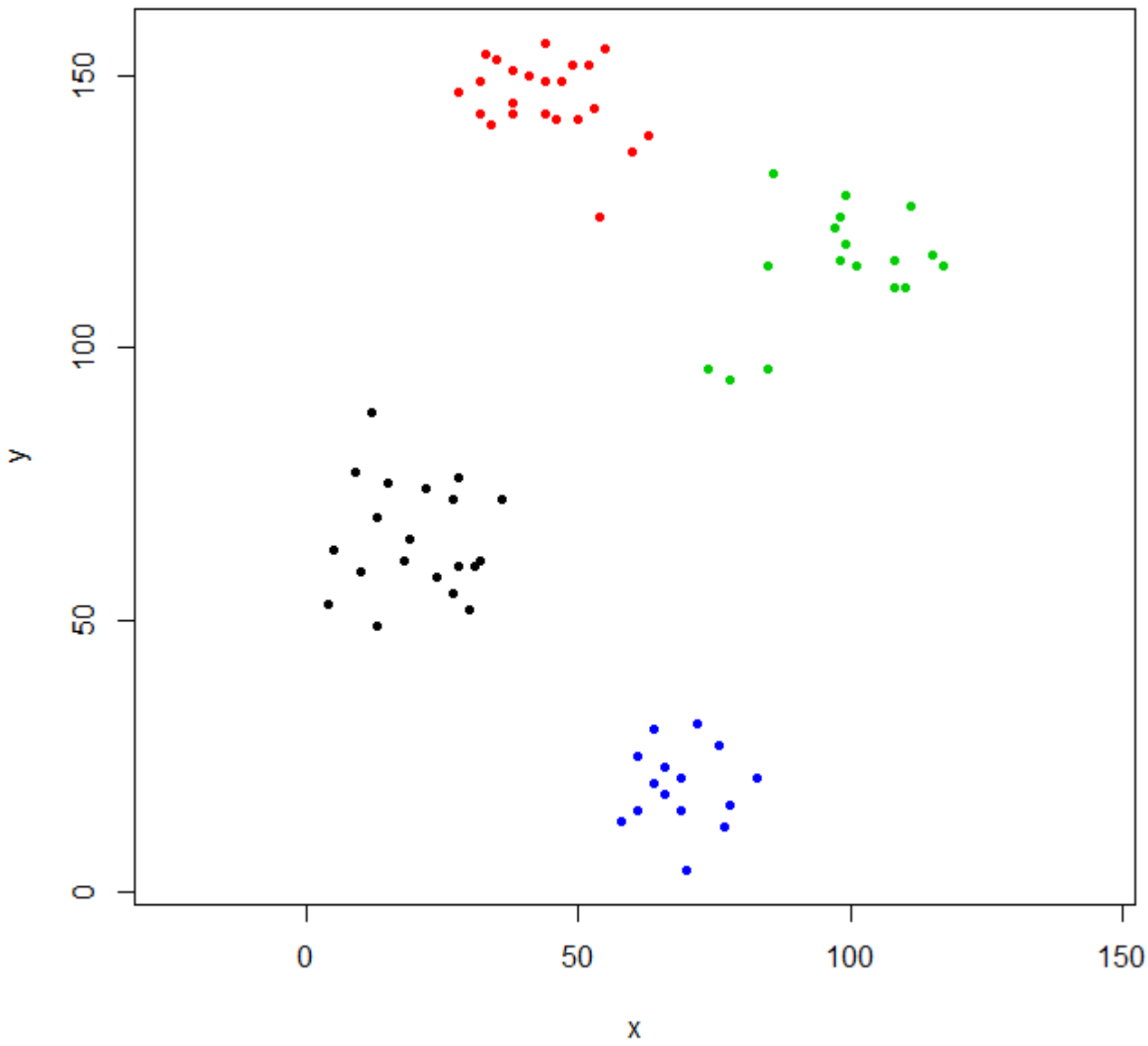
```
rhc_def_4 = cutree(ruspini_hc_defaults,k=4)
plot(ruspini, pch=20, asp=1, col=rhc_def_4)
```



This clustering is a little odd. We can get a better clustering by scaling the data first.

```
scaled_ruspini_hc_defaults = hclust(dist(scale(ruspini)))
srhc_def_4 = cutree(scaled_ruspini_hc_defaults,4)
plot(ruspini, pch=20, asp=1, col=srhc_def_4)
```


www.dbooks.org



The default dissimilarity measure for comparing clusters is "complete". You can specify a different measure with the method parameter.

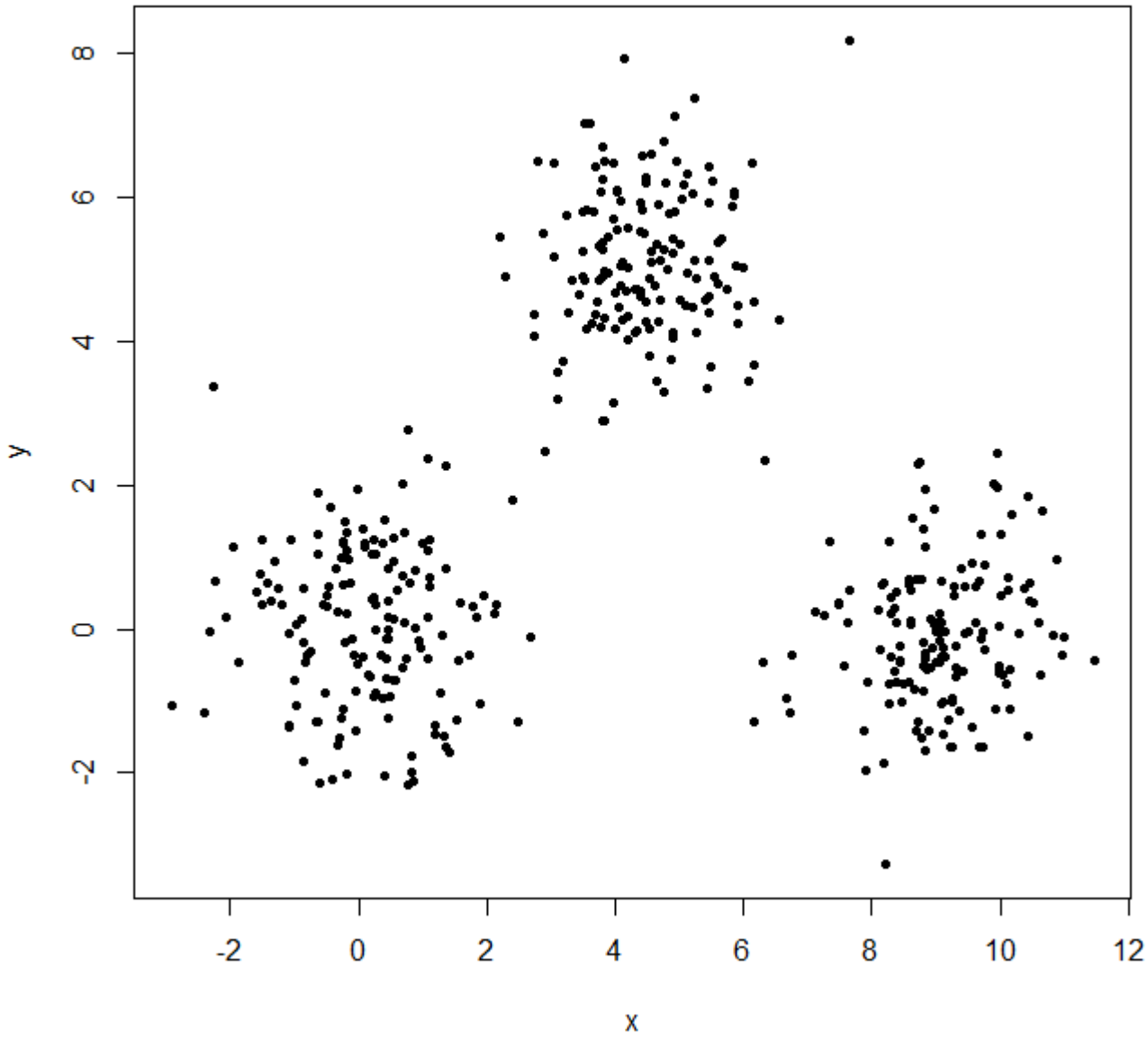
```
ruspini_hc_single = hclust(dist(ruspini), method="single")
```

Example 2 - hclust and outliers

With hierarchical clustering, outliers often show up as one-point clusters.

Generate three Gaussian distributions to illustrate the effect of outliers.

```
set.seed(656)
x = c(rnorm(150, 0, 1), rnorm(150,9,1), rnorm(150,4.5,1))
y = c(rnorm(150, 0, 1), rnorm(150,0,1), rnorm(150,5,1))
XYdf = data.frame(x,y)
plot(XYdf, pch=20)
```



Build the cluster structure, split it into three cluster.

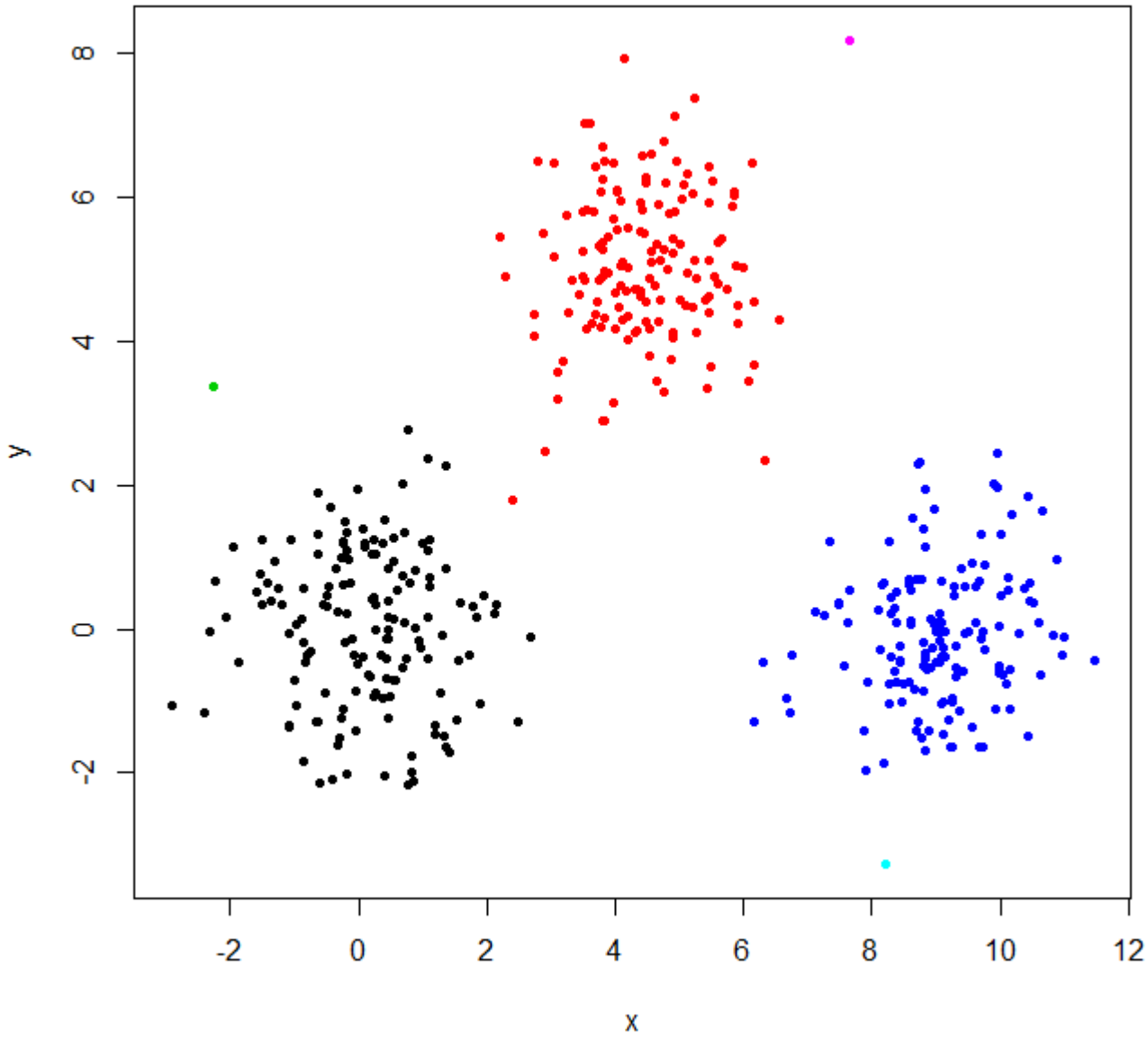
```
XY_sing = hclust(dist(XYdf), method="single")
XYs3 = cutree(XY_sing,k=3)
table(XYs3)
XYs3
  1   2   3
448   1   1
```

hclust found two outliers and put everything else into one big cluster. To get the "real" clusters, you may need to set k higher.

```
XYs6 = cutree(XY_sing,k=6)
table(XYs6)
XYs6
  1   2   3   4   5   6
```


www.dbooks.org

```
148 150   1 149   1   1
plot(XYdf, pch=20, col=XYs6)
```



This StackOverflow post has some guidance on how to pick the number of clusters, but be aware of this behavior in hierarchical clustering.

Read Hierarchical clustering with hclust online: https://riptutorial.com/r/topic/8084/hierarchical-clustering-with-hclust

Chapter 48: Hierarchical Linear Modeling

Examples

basic model fitting

apologies: *since I don't know of a channel for discussing/providing feedback on requests for improvement, I'm going to put my question here. Please feel free to point out a better place for this!* @DataTx states that this is "completely unclear, incomplete, or has severe formatting problems". Since I don't see any big formatting problems (:-)), a little bit more guidance about what's expected here for improving clarity or completeness, and why what's here is unsalvageable, would be useful.

The primary packages for fitting hierarchical (alternatively "mixed" or "multilevel") linear models in R are `nlme` (older) and `lme4` (newer). These packages differ in many minor ways but should generally result in very similar fitted models.

```
library(nlme)
library(lme4)
m1.nlme <- lme(Reaction~Days,random=~Days|Subject,data=sleepstudy,method="REML")
m1.lme4 <- lmer(Reaction~Days+(Days|Subject),data=sleepstudy,REML=TRUE)
all.equal(fixef(m1.nlme),fixef(m1.lme4))
## [1] TRUE
```

Differences to consider:

- formula syntax is slightly different
- `nlme` is (still) somewhat better documented (e.g. Pinheiro and Bates 2000 *Mixed-effects models in S-PLUS*; however, see Bates *et al.* 2015 *Journal of Statistical Software*/ `vignette("lmer",package="lme4")` for `lme4`)
- `lme4` is faster and allows easier fitting of crossed random effects
- `nlme` provides p-values for linear mixed models out of the box, `lme4` requires add-on packages such as `lmerTest` or `afex`
- `nlme` allows modeling of heteroscedasticity or residual correlations (in space/time/phylogeny)

The unofficial GLMM FAQ provides more information, although it is focused on *generalized* linear mixed models (GLMMs).

Read Hierarchical Linear Modeling online: https://riptutorial.com/r/topic/3460/hierarchical-linear-modeling

www.dbooks.org

Chapter 49: I/O for database tables

Remarks

Specialized packages

- RMySQL
- RODBC

Examples

Reading Data from MySQL Databases

General

Using the package RMySQL we can easily query MySQL as well as MariaDB databases and store the result in an R dataframe:

```
library(RMySQL)

mydb <- dbConnect(MySQL(), user='user', password='password', dbname='dbname',host='127.0.0.1')

queryString <- "SELECT * FROM table1 t1 JOIN table2 t2 on t1.id=t2.id"
query <- dbSendQuery(mydb, queryString)
data <- fetch(query, n=-1) # n=-1 to return all results
```

Using limits

It is also possible to define a limit, e.g. getting only the first 100,000 rows. In order to do so, just change the SQL query regarding the desired limit. The mentioned package will consider these options. Example:

```
queryString <- "SELECT * FROM table1 limit 100000"
```

Reading Data from MongoDB Databases

In order to load data from a MongoDB database into an R dataframe, use the library MongoLite:

```
# Use MongoLite library:
#install.packages("mongolite")
library(jsonlite)
library(mongolite)
```

```
# Connect to the database and the desired collection as root:
db <- mongo(collection = "Tweets", db = "TweetCollector", url =
"mongodb://USERNAME:PASSWORD@HOSTNAME")

# Read the desired documents i.e. Tweets inside one dataframe:
documents <- db$find(limit = 100000, skip = 0, fields = '{ "_id" : false, "Text" : true }')
```

The code connects to the server `HOSTNAME` as `USERNAME` with `PASSWORD`, tries to open the database `TweetCollector` and read the collection `Tweets`. The query tries to read the field i.e. column `Text`.

The results is a dataframe with columns as the yielded data set. In case of this example, the dataframe contains the column `Text`, e.g. `documents$Text`.

Read I/O for database tables online: https://riptutorial.com/r/topic/5537/i-o-for-database-tables

www.dbooks.org

Chapter 50: I/O for foreign tables (Excel, SAS, SPSS, Stata)

Examples

Importing data with rio

A very simple way to import data from many common file formats is with rio. This package provides a function `import()` that wraps many commonly used data import functions, thereby providing a standard interface. It works simply by passing a file name or URL to `import()`:

```
import("example.csv")        # comma-separated values
import("example.tsv")        # tab-separated values
import("example.dta")        # Stata
import("example.sav")        # SPSS
import("example.sas7bdat")   # SAS
import("example.xlsx")       # Excel
```

`import()` can also read from compressed directories, URLs (HTTP or HTTPS), and the clipboard. A comprehensive list of all supported file formats is available on the rio package github repository.

It is even possible to specify some further parameters related to the specific file format you are trying to read, passing them directly within the `import()` function:

```
import("example.csv", format = ",") #for csv file where comma is used as separator
import("example.csv", format = ";") #for csv file where semicolon is used as separator
```

Importing Excel files

There are several R packages to read excel files, each of which using different languages or resources, as summarized in the following table:

R package	Uses
xlsx	Java
XLconnect	Java
openxlsx	C++
readxl	C++
RODBC	ODBC
gdata	Perl

For the packages that use Java or ODBC it is important to know details about your system because you may have compatibility issues depending on your R version and OS. For instance, if you are using R 64 bits then you also must have Java 64 bits to use `xlsx` or `XLconnect`.

Some examples of reading excel files with each package are provided below. Note that many of the packages have the same or very similar function names. Therefore, it is useful to state the package explicitly, like `package::function`. The package `openxlsx` requires prior installation of RTools.

Reading excel files with the xlsx package

```
library(xlsx)
```

The index or name of the sheet is required to import.

```
xlsx::read.xlsx("Book1.xlsx", sheetIndex=1)

xlsx::read.xlsx("Book1.xlsx", sheetName="Sheet1")
```

Reading Excel files with the XLconnect package

```
library(XLConnect)
wb <- XLConnect::loadWorkbook("Book1.xlsx")

# Either, if Book1.xlsx has a sheet called "Sheet1":
sheet1 <- XLConnect::readWorksheet(wb, "Sheet1")
# Or, more generally, just get the first sheet in Book1.xlsx:
sheet1 <- XLConnect::readWorksheet(wb, getSheets(wb)[1])
```

`XLConnect` automatically imports the pre-defined Excel cell-styles embedded in `Book1.xlsx`. This is useful when you wish to format your workbook object and export a perfectly formatted Excel document. Firstly, you will need to create the desired cell formats in `Book1.xlsx` and save them, for example, as `myHeader`, `myBody` and `myPcts`. Then, after loading the workbook in `R` (see above):

```
Headerstyle <- XLConnect::getCellStyle(wb, "myHeader")
Bodystyle <- XLConnect::getCellStyle(wb, "myBody")
Pctsstyle <- XLConnect::getCellStyle(wb, "myPcts")
```

The cell styles are now saved in your `R` environment. In order to assign the cell styles to certain ranges of your data, you need to define the range and then assign the style:

```
Headerrange <- expand.grid(row = 1, col = 1:8)
Bodyrange <- expand.grid(row = 2:6, col = c(1:5, 8))
Pctrange <- expand.grid(row = 2:6, col = c(6, 7))
```


www.dbooks.org

```
XLConnect::setCellStyle(wb, sheet = "sheet1", row = Headerrange$row,
             col = Headerrange$col, cellstyle = Headerstyle)
XLConnect::setCellStyle(wb, sheet = "sheet1", row = Bodyrange$row,
             col = Bodyrange$col, cellstyle = Bodystyle)
XLConnect::setCellStyle(wb, sheet = "sheet1", row = Pctrange$row,
             col = Pctrange$col, cellstyle = Pctsstyle)
```

Note that `XLConnect` is easy, but can become extremely slow in formatting. A much faster, but more cumbersome formatting option is offered by `openxlsx`.

Reading excel files with the openxlsx package

Excel files can be imported with package `openxlsx`

```
library(openxlsx)

openxlsx::read.xlsx("spreadsheet1.xlsx", colNames=TRUE, rowNames=TRUE)

#colNames: If TRUE, the first row of data will be used as column names.
#rowNames: If TRUE, first column of data will be used as row names.
```

The sheet, which should be read into R can be selected either by providing its position in the `sheet` argument:

```
openxlsx::read.xlsx("spreadsheet1.xlsx", sheet = 1)
```

or by declaring its name:

```
openxlsx::read.xlsx("spreadsheet1.xlsx", sheet = "Sheet1")
```

Additionally, `openxlsx` can detect date columns in a read sheet. In order to allow automatic detection of dates, an argument `detectDates` should be set to `TRUE`:

```
openxlsx::read.xlsx("spreadsheet1.xlsx", sheet = "Sheet1", detectDates= TRUE)
```

Reading excel files with the readxl package

Excel files can be imported as a data frame into `R` using the `readxl` package.

```
library(readxl)
```

It can read both `.xls` and `.xlsx` files.

```
readxl::read_excel("spreadsheet1.xls")
```

```
readxl::read_excel("spreadsheet2.xlsx")
```

The sheet to be imported can be specified by number or name.

```
readxl::read_excel("spreadsheet.xls", sheet = 1)
readxl::read_excel("spreadsheet.xls", sheet = "summary")
```

The argument `col_names = TRUE` sets the first row as the column names.

```
 readxl::read_excel("spreadsheet.xls", sheet = 1, col_names = TRUE)
```

The argument `col_types` can be used to specify the column types in the data as a vector.

```
readxl::read_excel("spreadsheet.xls", sheet = 1, col_names = TRUE,
                   col_types = c("text", "date", "numeric", "numeric"))
```

Reading excel files with the RODBC package

Excel files can be read using the ODBC Excel Driver that interfaces with Windows' Access Database Engine (ACE), formerly JET. With the RODBC package, R can connect to this driver and directly query workbooks. Worksheets are assumed to maintain column headers in first row with data in organized columns of similar types. **NOTE:** This approach is limited to only Windows/PC machines as JET/ACE are installed .dll files and not available on other operating systems.

```
library(RODBC)

xlconn <- odbcDriverConnect('Driver={Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)};
                            DBQ=C:\\Path\\To\\Workbook.xlsx')

df <- sqlQuery(xlconn, "SELECT * FROM [SheetName$]")
close(xlconn)
```

Connecting with an SQL engine in this approach, Excel worksheets can be queried similar to database tables including `JOIN` and `UNION` operations. Syntax follows the JET/ACE SQL dialect. **NOTE:** Only data access DML statements, specifically `SELECT` can be run on workbooks, considered not updateable queries.

```
joindf <-  sqlQuery(xlconn, "SELECT t1.*, t2.* FROM [Sheet1$] t1
                            INNER JOIN [Sheet2$] t2
                            ON t1.[ID] = t2.[ID]")

uniondf <-  sqlQuery(xlconn, "SELECT * FROM [Sheet1$]
                             UNION
                             SELECT * FROM [Sheet2$]")
```

Even other workbooks can be queried from the same ODBC channel pointing to a current workbook:

www.dbooks.org

```
otherwkbkdf <- sqlQuery(xlconn, "SELECT * FROM
                                 [Excel 12.0 Xml;HDR=Yes;
                                 Database=C:\\Path\\To\\Other\\Workbook.xlsx].[Sheet1$];")
```

Reading excel files with the gdata package

example here

Read and write Stata, SPSS and SAS files

The packages `foreign` and `haven` can be used to import and export files from a variety of other statistical packages like Stata, SPSS and SAS and related software. There is a `read` function for each of the supported data types to import the files.

```
# loading the packages
library(foreign)
library(haven)
library(readstata13)
library(Hmisc)
```

Some examples for the most common data types:

```
# reading Stata files with `foreign`
read.dta("path\to\your\data")
# reading Stata files with `haven`
read_dta("path\to\your\data")
```

The `foreign` package can read in stata (.dta) files for versions of Stata 7-12. According to the development page, the `read.dta` is more or less frozen and will not be updated for reading in versions 13+. For more recent versions of Stata, you can use either the `readstata13` package or `haven`. For `readstata13`, the files are

```
# reading recent Stata (13+) files with `readstata13`
read.dta13("path\to\your\data")
```

For reading in SPSS and SAS files

```
# reading SPSS files with `foreign`
read.spss("path\to\your\data.sav", to.data.frame = TRUE)
# reading SPSS files with `haven`
read_spss("path\to\your\data.sav")
read_sav("path\to\your\data.sav")
read_por("path\to\your\data.por")

# reading SAS files with `foreign`
read.ssd("path\to\your\data")
# reading SAS files with `haven`
read_sas("path\to\your\data")
# reading native SAS files with `Hmisc`
sas.get("path\to\your\data")   #requires access to saslib
# Reading SA XPORT format ( *.XPT ) files
```

```
sasxport.get("path\to\your\data.xpt")  # does not require access to SAS executable
```

The `SAScii` package provides functions that will accept SAS SET import code and construct a text file that can be processed with `read.fwf`. It has proved very robust for import of large public-released datasets. Support is at https://github.com/ajdamico/SAScii

To export data frames to other statistical packages you can use the write functions `write.foreign()`. This will write 2 files, one containing the data and one containing instructions the other package needs to read the data.

```
# writing to Stata, SPSS or SAS files with `foreign`
write.foreign(dataframe, datafile, codefile,
              package = c("SPSS", "Stata", "SAS"), ...)
write.foreign(dataframe, "path\to\data\file", "path\to\instruction\file", package = "Stata")

# writing to Stata files with `foreign`
write.dta(dataframe, "file", version = 7L,
          convert.dates = TRUE, tz = "GMT",
          convert.factors = c("labels", "string", "numeric", "codes"))

# writing to Stata files with `haven`
write_dta(dataframe, "path\to\your\data")

# writing to Stata files with `readstata13`
save.dta13(dataframe, file, data.label = NULL, time.stamp = TRUE,
  convert.factors = TRUE, convert.dates = TRUE, tz = "GMT",
  add.rownames = FALSE, compress = FALSE, version = 117,
  convert.underscore = FALSE)

# writing to SPSS files with `haven`
write_sav(dataframe, "path\to\your\data")
```

File stored by the SPSS can also be read with `read.spss` in this way:

```
 foreign::read.spss('data.sav', to.data.frame=TRUE, use.value.labels=FALSE,
                    use.missings=TRUE, reencode='UTF-8')
# to.data.frame if TRUE: return a data frame
# use.value.labels if TRUE: convert variables with value labels into R factors with those
levels
# use.missings if TRUE: information on user-defined missing values will used to set the
corresponding values to NA.
# reencode character strings will be re-encoded to the current locale. The default, NA, means
to do so in a UTF-8 locale, only.
```

Import or Export of Feather file

Feather is an implementation of Apache Arrow designed to store data frames in a language agnostic manner while maintaining metadata (e.g. date classes), increasing interoperability between Python and R. Reading a feather file will produce a tibble, not a standard data.frame.

```
library(feather)

path <- "filename.feather"
df <- mtcars
```


www.dbooks.org

```
write_feather(df, path)

df2 <- read_feather(path)

head(df2)
##  A tibble: 6 x 11
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  21.0     6   160   110  3.90 2.620 16.46     0     1     4     4
## 2  21.0     6   160   110  3.90 2.875 17.02     0     1     4     4
## 3  22.8     4   108    93  3.85 2.320 18.61     1     1     4     1
## 4  21.4     6   258   110  3.08 3.215 19.44     1     0     3     1
## 5  18.7     8   360   175  3.15 3.440 17.02     0     0     3     2
## 6  18.1     6   225   105  2.76 3.460 20.22     1     0     3     1

head(df)
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

The current documentation contains this warning:

> Note to users: Feather should be treated as alpha software. In particular, the file format is likely to evolve over the coming year. Do not use Feather for long-term data storage.

Read I/O for foreign tables (Excel, SAS, SPSS, Stata) online: https://riptutorial.com/r/topic/5536/i-o-for-foreign-tables--excel--sas--spss--stata-

Chapter 51: I/O for geographic data (shapefiles, etc.)

Introduction

See also Introduction to Geographical Maps and Input and Output

Examples

Import and Export Shapefiles

With the `rgdal` package it is possible to import and export shapfiles with R. The function `readOGR` can be used to imports shapfiles. If you want to import a file from e.g. ArcGIS the first argument `dsn` is the path to the folder which contains the shapefile. `layer` is the name of the shapefile without the file ending (just `map` and not `map.shp`).

```
library(rgdal)
readOGR(dsn = "path\to\the\folder\containing\the\shapefile", layer = "map")
```

To export a shapefile use the`writeOGR` function. The first argument is the spatial object produced in R. `dsn` and `layer` are the same as above. The obligatory 4. argument is the driver used to generate the shapefile. The function `ogrDrivers()` lists all available drivers. If you want to export a shapfile to ArcGis or QGis you could use `driver = "ESRI Shapefile"`.

```
writeOGR(Rmap, dsn = "path\to\the\folder\containing\the\shapefile", layer = "map",
         driver = "ESRI Shapefile" )
```

`tmap` package has a very convenient function `read_shape()`, which is a wrapper for `rgdal::reagOGR()`. The `read_shape()` function simplifies the process of importing a shapefile a lot. On the downside, `tmap` is quite heavy.

Read I/O for geographic data (shapefiles, etc.) online: https://riptutorial.com/r/topic/5538/i-o-for-geographic-data--shapefiles--etc--

www.dbooks.org

Chapter 52: I/O for raster images

Introduction

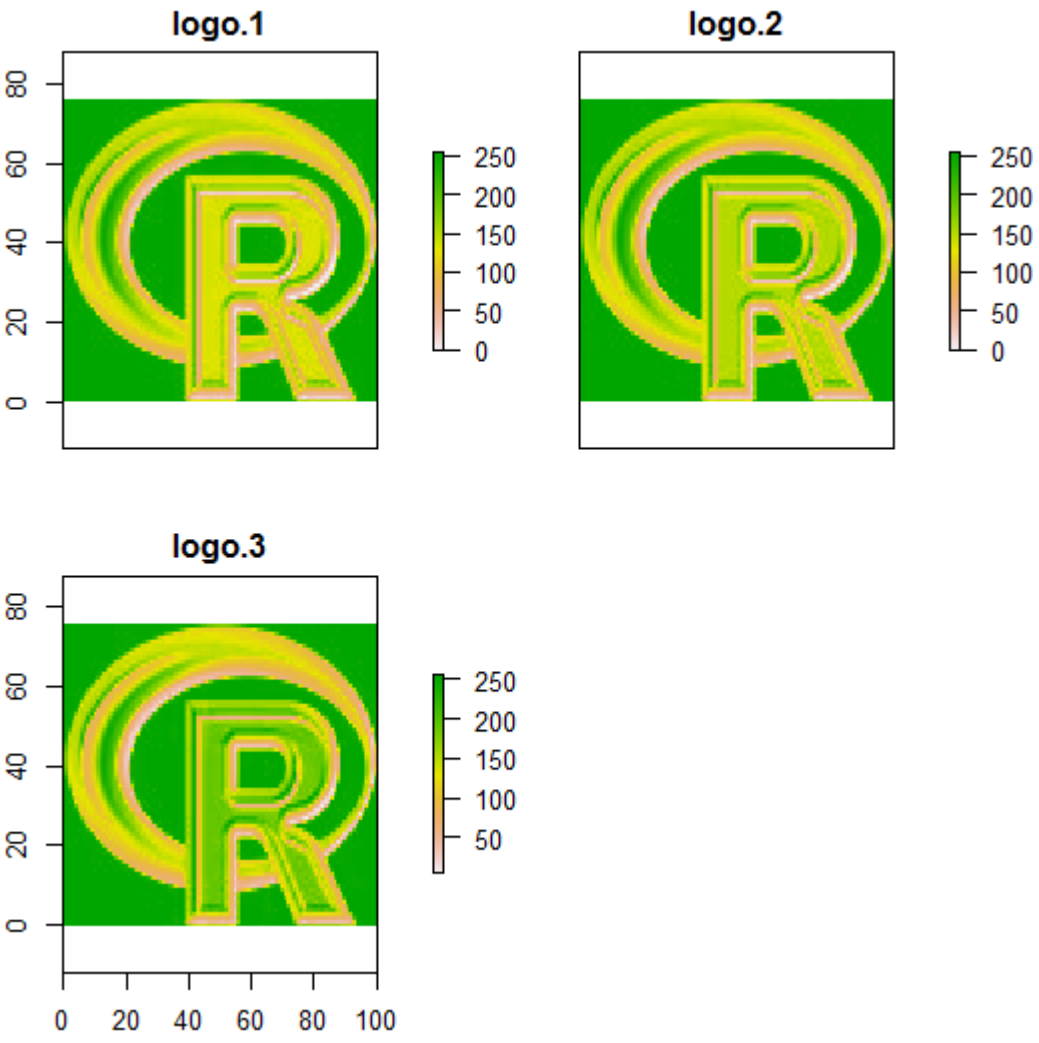
See also Raster and Image Analysis and Input and Output

Examples

Load a multilayer raster

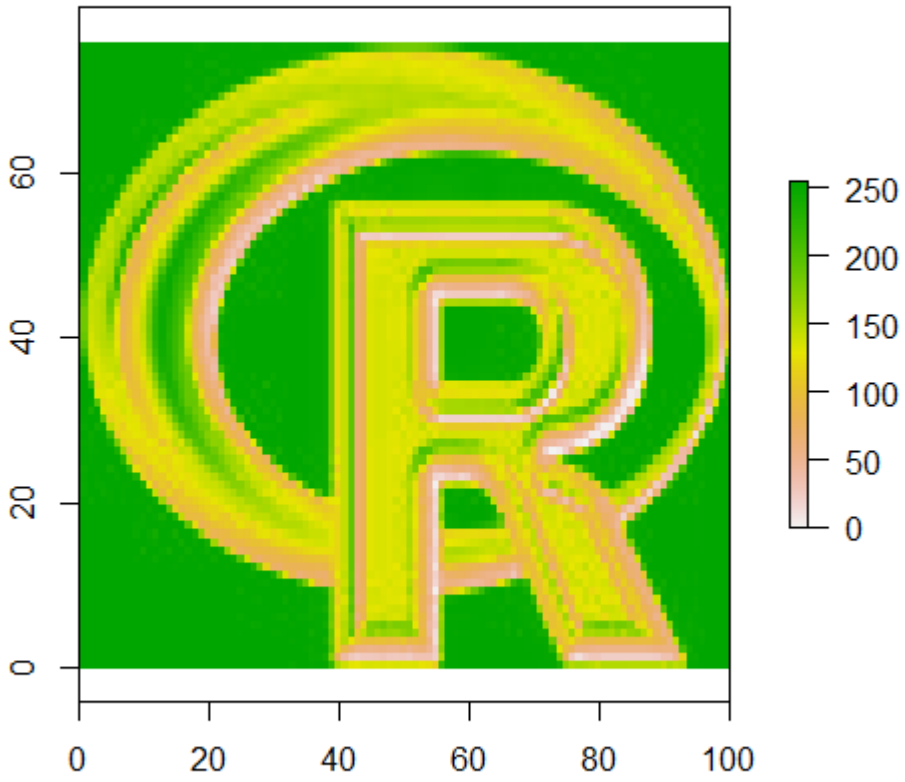
The R-Logo is a multilayer raster file (red, green, blue)

```
library(raster)
r <- stack("C:/Program Files/R/R-3.2.3/doc/html/logo.jpg")
plot(r)
```



The individual layers of the `RasterStack` object can be adressed by `[[`.

```
plot(r[[1]])
```



Read I/O for raster images online: https://riptutorial.com/r/topic/5539/i-o-for-raster-images

www.dbooks.org

Chapter 53: I/O for R's binary format

Examples

Rds and RData (Rda) files

`.rds` and `.Rdata` (also known as `.rda`) files can be used to store R objects in a format native to R. There are multiple advantages of saving this way when contrasted with non-native storage approaches, e.g. `write.table`:

- It is faster to restore the data to R
- It keeps R specific information encoded in the data (e.g., attributes, variable types, etc).

`saveRDS`/`readRDS` only handle a single R object. However, they are more flexible than the multi-object storage approach in that the object name of the restored object need not be the same as the object name when the object was stored.

Using an .rds file, for example, saving the `iris` dataset we would use:

```
saveRDS(object = iris, file = "my_data_frame.rds")
```

To load it data back in:

```
iris2 <- readRDS(file = "my_data_frame.rds")
```

To save a multiple objects we can use `save()` and output as `.Rdata`.

Example, to save 2 dataframes: iris and cars

```
save(iris, cars, file = "myIrisAndCarsData.Rdata")
```

To load:

```
load("myIrisAndCarsData.Rdata")
```

Enviromments

The functions `save` and `load` allow us to specify the environment where the object will be hosted:

```
save(iris, cars, file = "myIrisAndCarsData.Rdata", envir = foo  <- new.env())
load("myIrisAndCarsData.Rdata", envir = foo)
foo$cars

save(iris, cars, file = "myIrisAndCarsData.Rdata", envir = foo  <- new.env())
load("myIrisAndCarsData.Rdata", envir = foo)
```

```
foo$cars
```

Read I/O for R's binary format online: https://riptutorial.com/r/topic/5540/i-o-for-r-s-binary-format

www.dbooks.org

Chapter 54: Implement State Machine Pattern using S4 Class

Introduction

Finite States Machine concepts are usually implemented under Object Oriented Programming (OOP) languages, for example using Java language, based on the State pattern defined in GOF (refers to the book: "Design Patterns").

R provides several mechanisms to simulate the OO paradigm, let's apply S4 Object System for implementing this pattern.

Examples

Parsing Lines using State Machine

Let's apply the State Machine pattern for parsing lines with the specific pattern using S4 Class feature from R.

PROBLEM ENUNCIATION

We need to parse a file where each line provides information about a person, using a delimiter (`";"`), but some information provided is optional, and instead of providing an empty field, it is missing. On each line we can have the following information: `Name;[Address;]Phone`. Where the address information is optional, sometimes we have it and sometimes don't, for example:

```
GREGORY BROWN; 25 NE 25TH; +1-786-987-6543
DAVID SMITH;786-123-4567
ALAN PEREZ; 25 SE 50TH; +1-786-987-5553
```

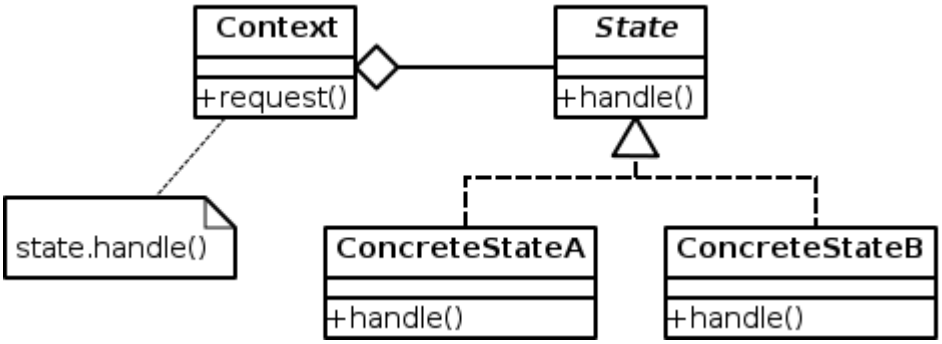
The second line does not provide address information. Therefore the number of delimiters may be deferent like in this case with one delimiter and for the other lines two delimiters. Because the number of delimiters may vary, one way to atack this problem is to recognize the presence or not of a given field based on its pattern. In such case we can use a regular expression for identifying such patterns. For example:

- **Name**: `"^([A-Z]'?\\s+)* *[A-Z]+(\\s+[A-Z]{1,2}\\.?,? +)*[A-Z]+((-|\\s+)[A-Z]+)*$"`. For example: `RAFAEL REAL, DAVID R. SMITH, ERNESTO PEREZ GONZALEZ, O' CONNOR BROWN, LUIS PEREZ-MENA`, etc.
- **Address**: `"^\\s[0-9]{1,4}(\\s+[A-Z]{1,2}[0-9]{1,2}[A-Z]{1,2}|[A-Z\\s0-9]+)$"`. For example: `11020 LE JEUNE ROAD, 87 SW 27TH`. For the sake of simplicity we don't include here the zipcode, city, state, but I can be included in this field or adding additional fields.
- **Phone**: `"^\\s*(\\+1(-|\\s+))*[0-9]{3}(-|\\s+)[0-9]{3}(-|\\s+)[0-9]{4}$"`. For example: `305-123-4567, 305 123 4567, +1-786-123-4567`.

Notes:

- I am considering the most common pattern of US addresses and phones, it can be easy extended to consider more general situations.
- In R the sign `"\"` has special meaning for character variables, therefore we need to escape it.
- In order to simplify the process of defining regular expressions a good recommendation is to use the following web page: regex101.com, so you can play with it, with a given example, until you get the expected result for all possible combinations.

The idea is to identify each line field based on previously defined patterns. The State pattern define the following entities (classes) that collaborate to control the specific behavior (The State Pattern is a behavior pattern):



Let's describe each element considering the context of our problem:

- `Context`: Stores the context information of the parsing process, i.e. the current state and handles the entire State Machine Process. For each state, an action is executed (`handle()`), but the context delegates it, based on the state, on the action method defined for a particular state (`handle()` from `State` class). It defines the interface of interest to clients. Our `Context` class can be defined like this:
  - Attributes: `state`
  - Methods: `handle()`, ...
- `State`: The abstract class that represents any state of the State Machine. It defines an interface for encapsulating the behavior associated with a particular state of the context. It can be defined like this:
  - Attributes: `name, pattern`
  - Methods: `doAction()`, `isState` (using `pattern` attribute verify whether the input argument belong to this state pattern or not), …
- `Concrete States` (state sub-classes): Each subclass of the class `State` that implements a behavior associated with a state of the `Context`. Our sub-classes are: `InitState`, `NameState`, `AddressState`, `PhoneState`. Such classes just implements the generic method using the specific logic for such states. No additional attributes are required.

Note: It is a matter of preference how to name the method that carries out the action, `handle()`, `doAction()` or `goNext()`. The method name `doAction()` can be the same for both classes (`State`or `Context`) we preferred to name as `handle()` in the `Context` class for avoiding a confusion when defining two generic methods with the same input arguments, but different class.

www.dbooks.org

PERSON CLASS

Using the S4 syntax we can define a Person class like this:

```
setClass(Class = "Person",
    slots = c(name = "character", address = "character", phone = "character")
)
```

It is a good recommendation to initialize the class attributes. The `setClass` documentation suggests using a generic method labeled as `"initialize"`, instead of using deprecated attributes such as: `prototype, representation`.

```
setMethod("initialize", "Person",
  definition = function(.Object, name = NA_character_,
    address = NA_character_, phone = NA_character_) {
        .Object@name <- name
        .Object@address <- address
        .Object@phone <- phone
        .Object
    }
)
```

Because the initialize method is already a standard generic method of package `methods`, we need to respect the original argument definition. We can verify it typing on R prompt:

```
> initialize
```

It returns the entire function definition, you can see at the top who the function is defined like:

```
function (.Object, ...) {...}
```

Therefore when we use `setMethod` we need to follow *exaclty* the same syntax (`.Object`).

Another existing generic method is `show`, it is equivalent `toString()` method from Java and it is a good idea to have a specific implementation for class domain:

```
setMethod("show", signature = "Person",
  definition = function(object) {
      info <- sprintf("%s@[name='%s', address='%s', phone='%s']",
        class(object), object@name, object@address, object@phone)
      cat(info)
      invisible(NULL)
  }
)
```

Note: We use the same convention as in the default `toString()` Java implementation.

Let's say we want to save the parsed information (a list of `Person` objects) into a dataset, then we should be able first to convert a list of objects to into something the R can transform (for example coerce the object as a list). We can define the following additional method (for more detail about this see the post)

```
setGeneric(name = "as.list", signature = c('x'),
    def = function(x) standardGeneric("as.list"))

# Suggestion taken from here:
# http://stackoverflow.com/questions/30386009/how-to-extend-as-list-in-a-canonical-way-to-s4-
objects
setMethod("as.list", signature = "Person",
    definition = function(x) {
        mapply(function(y) {
        #apply as.list if the slot is again an user-defined object
        #therefore, as.list gets applied recursively
        if (inherits(slot(x,y),"Person")) {
          as.list(slot(x,y))
        } else {
          #otherwise just return the slot
          slot(x,y)
        }
      },
        slotNames(class(x)),
        SIMPLIFY=FALSE)
    }
)
```

R does not provide a sugar syntax for OO because the language was initially conceived to provide valuable functions for Statisticians. Therefore each user method requires two parts: 1) the Definition part (via `setGeneric`) and 2) the implementation part (via `setMethod`). Like in the above example.

STATE CLASS

Following S4 syntax, let's define the abstract `State` class.

```
setClass(Class = "State", slots = c(name = "character", pattern = "character"))

setMethod("initialize", "State",
  definition = function(.Object, name = NA_character_, pattern = NA_character_) {
      .Object@name <- name
      .Object@pattern <- pattern
      .Object
  }
)

setMethod("show", signature = "State",
  definition = function(object) {
      info <- sprintf("%s@[name='%s', pattern='%s']", class(object),
          object@name, object@pattern)
      cat(info)
      invisible(NULL)
  }
)

setGeneric(name = "isState", signature = c('obj', 'input'),
    def = function(obj, input) standardGeneric("isState"))

setGeneric(name = "doAction", signature = c('obj', 'input', 'context'),
    def = function(obj, input, context) standardGeneric("doAction"))
```

Every sub-class from `State` will have associated a `name` and `pattern`, but also a way to identify

www.dbooks.org

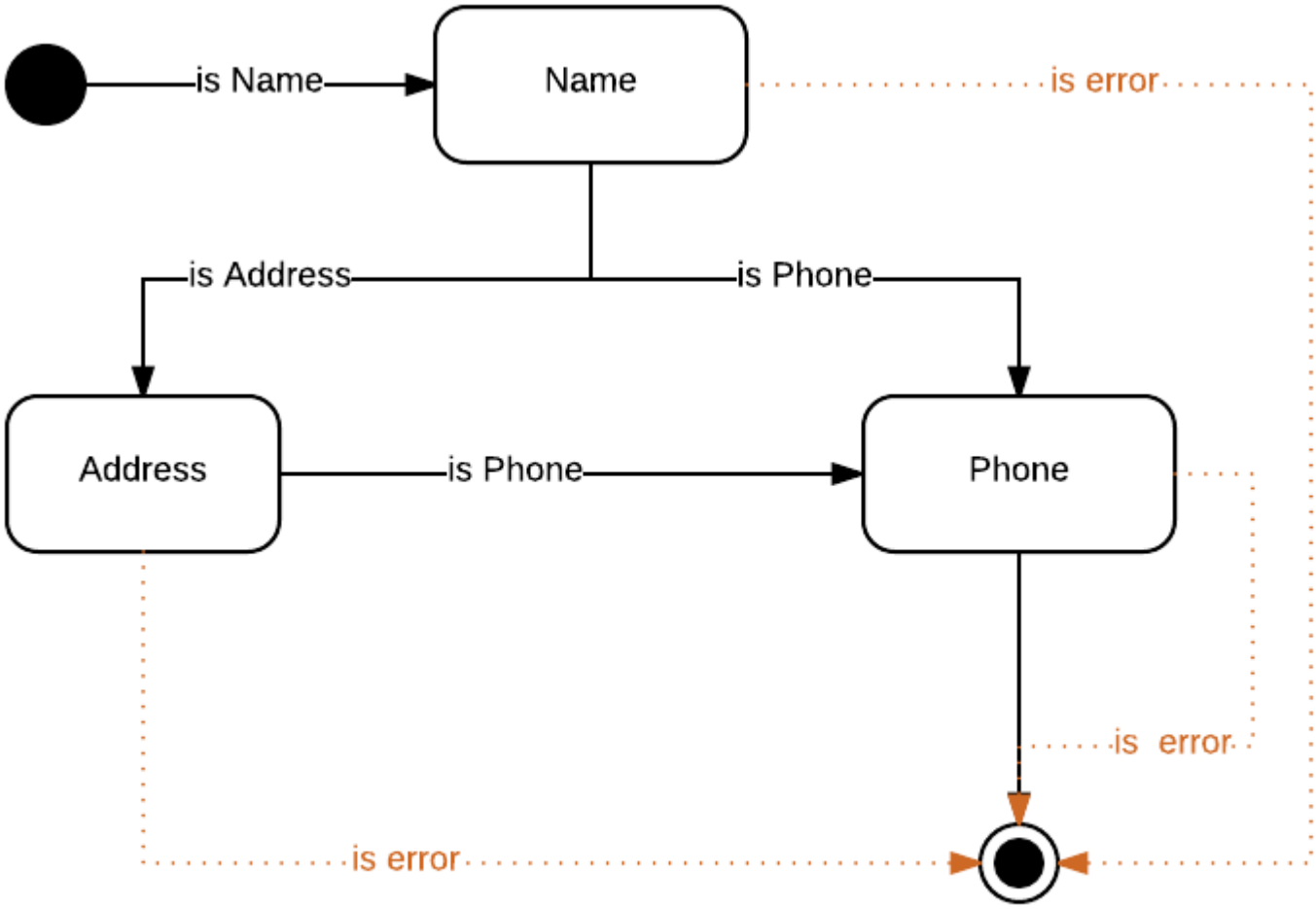
whether a given input belongs to this state or not (`isState()` method), and also implement the corresponding actions for this state (`doAction()` method).

In order to understand the process, let's define the transition matrix for each state based on the input received:

Input/Current State	Init	Name	Address	Phone
Name	Name			
Address		Address		
Phone		Phone	Phone	
End				End

Note: The cell `[row, col]=[i,j]` represents the destination state for the current state `j`, when it receives the input `i`.

It means that under the state Name it can receive two inputs: an address or a phone number. Another way to represents the transaction table is using the following UML State Machine diagram:



is error: when the input argument has an invalid pattern

Let's implement each particular state as a sub-state of the class `State`

STATE SUB-CLASSES

Init State:

The initial state will be implemented via the following class:

```
setClass("InitState", contains = "State")

setMethod("initialize", "InitState",
  definition = function(.Object, name = "init", pattern = NA_character_) {
      .Object@name <- name
      .Object@pattern <- pattern
      .Object
  }
)

setMethod("show", signature = "InitState",
  definition = function(object) {
      callNextMethod()
  }
```


www.dbooks.org

```
)
```

In R to indicate a class is a sub-class of other class is using the attribute `contains` and indicating the class name of the parent class.

Because the sub-classes just implement the generic methods, without adding additional attributes, then the `show` method, just call the equivalent method from the upper class (via method: `callNextMethod()`)

The initial state does not have associated a pattern, it just represents the beginning of the process, then we initialize the class with an `NA` value.

Now lets to implement the generic methods from the `State` class:

```
setMethod(f = "isState", signature = "InitState",
  definition = function(obj, input) {
      nameState <- new("NameState")
      result <- isState(nameState, input)
      return(result)
  }
)
```

For this particular state (without `pattern`), the idea it just initializes the parsing process expecting the first field will be a `name`, otherwise it will be an error.

```
setMethod(f = "doAction", signature = "InitState",
    definition = function(obj, input, context) {
        nameState <- new("NameState")
        if (isState(nameState, input)) {
            person <- context@person
            person@name <- trimws(input)
            context@person <- person
            context@state <- nameState
        } else {
            msg <- sprintf("The input argument: '%s' cannot be identified", input)
            stop(msg)
        }
        return(context)
    }
)
```

The `doAction` method provides the transition and updates the context with the information extracted. Here we are accessing to context information via the `@-operator`. Instead, we can define `get/set` methods, to encapsulate this process (as it is mandated in OO best practices: encapsulation), but that would add four more methods per `get-set` without adding value for the purpose of this example.

It is a good recommendation in all `doAction` implementation, to add a safeguard when the input argument is not properly identified.

Name State

Here is the definition of this class definition:

```
setClass ("NameState", contains = "State")

setMethod("initialize","NameState",
  definition=function(.Object, name="name",
        pattern = "^([A-Z]'?\\s+)* *[A-Z]+(\\s+[A-Z]{1,2}\\.?,? +)*[A-Z]+((-|\\s+)[A-Z]+)*$")
{
        .Object@pattern <- pattern
        .Object@name <- name
        .Object
  }
)

setMethod("show", signature = "NameState",
  definition = function(object) {
      callNextMethod()
  }
)
```

We use the function `grepl` for verifying the input belongs to a given pattern.

```
setMethod(f="isState", signature="NameState",
  definition=function(obj, input) {
      result <- grepl(obj@pattern, input, perl=TRUE)
      return(result)
  }
)
```

Now we define the action to carry out for a given state:

```
setMethod(f = "doAction", signature = "NameState",
  definition=function(obj, input, context) {
      addressState <- new("AddressState")
      phoneState <- new("PhoneState")
      person <- context@person
      if (isState(addressState, input)) {
          person@address <- trimws(input)
          context@person <- person
          context@state <- addressState
      } else if (isState(phoneState, input)) {
          person@phone <- trimws(input)
          context@person <- person
          context@state <- phoneState
      } else {
          msg <- sprintf("The input argument: '%s' cannot be identified", input)
          stop(msg)
      }
      return(context)
  }
)
```

Here we consider to possible transitions: one for Address state and the other one for Phone state. In all cases we update the context information:

- The `person` information: `address` or `phone` with the input argument.
- The `state` of the process

The way to identify the state is to invoke the method: `isState()` for a particular state. We create a

www.dbooks.org

default specific states (`addressState, phoneState`) and then ask for a particular validation.

The logic for the other sub-classes (one per state) implementation is very similar.

Address State

```
setClass("AddressState", contains = "State")

setMethod("initialize", "AddressState",
  definition = function(.Object, name="address",
    pattern = "^\\s[0-9]{1,4}(\\s+[A-Z]{1,2}[0-9]{1,2}[A-Z]{1,2}|[A-Z\\s0-9]+)$") {
        .Object@pattern <- pattern
        .Object@name <- name
        .Object
    }
)

setMethod("show", signature = "AddressState",
  definition = function(object) {
      callNextMethod()
  }
)

setMethod(f="isState", signature="AddressState",
    definition=function(obj, input) {
        result <- grepl(obj@pattern, input, perl=TRUE)
        return(result)
    }
)

setMethod(f = "doAction", "AddressState",
    definition=function(obj, input, context) {
        phoneState <- new("PhoneState")
        if (isState(phoneState, input)) {
            person <- context@person
            person@phone <- trimws(input)
            context@person <- person
            context@state <- phoneState
        } else {
            msg <- sprintf("The input argument: '%s' cannot be identified", input)
            stop(msg)
        }
        return(context)
    }
)
```

Phone State

```
setClass("PhoneState", contains = "State")

setMethod("initialize", "PhoneState",
  definition = function(.Object, name = "phone",
    pattern = "^\\s*(\\+1(-|\\s+))*[0-9]{3}(-|\\s+)[0-9]{3}(-|\\s+)[0-9]{4}$") {
        .Object@pattern <- pattern
        .Object@name <- name
        .Object
    }
)
```

```
setMethod("show", signature = "PhoneState",
  definition = function(object) {
      callNextMethod()
  }
)

setMethod(f = "isState", signature = "PhoneState",
    definition = function(obj, input) {
        result <- grepl(obj@pattern, input, perl = TRUE)
        return(result)
    }
)
```

Here is where we add the person information into the list of `persons` of the `context`.

```
setMethod(f = "doAction", "PhoneState",
    definition = function(obj, input, context) {
        context <- addPerson(context, context@person)
        context@state <- new("InitState")
        return(context)
    }
)
```

CONTEXT CLASS

Now the lets to explain the `Context` class implementation. We can define it considering the following attributes:

```
setClass(Class = "Context",
     slots = c(state = "State", persons = "list", person = "Person")
)
```

Where

- `state`: The current state of the process
- `person`: The current person, it represents the information we have already parsed from the current line.
- `persons`: The list of parsed persons processed.

Note: Optionally, we can add a `name` to identify the context by name in case we are working with more than one parser type.

```
setMethod(f="initialize", signature="Context",
  definition = function(.Object) {
        .Object@state <- new("InitState")
        .Object@persons <- list()
        .Object@person <- new("Person")
        return(.Object)
    }
)

setMethod("show", signature = "Context",
  definition = function(object) {
      cat("An object of class ", class(object), "\n", sep = "")
      info <- sprintf("[state='%s', persons='%s', person='%s']", object@state,
```


www.dbooks.org

```
          toString(object@persons), object@person)
      cat(info)
      invisible(NULL)
  }
)

setGeneric(name = "handle", signature = c('obj', 'input', 'context'),
    def = function(obj, input, context) standardGeneric("handle"))

setGeneric(name = "addPerson", signature = c('obj', 'person'),
    def = function(obj, person) standardGeneric("addPerson"))

setGeneric(name = "parseLine", signature = c('obj', 's'),
    def = function(obj, s) standardGeneric("parseLine"))

setGeneric(name = "parseLines", signature = c('obj', 's'),
    def = function(obj, s) standardGeneric("parseLines"))

setGeneric(name = "as.df", signature = c('obj'),
    def = function(obj) standardGeneric("as.df"))
```

With such generic methods, we control the entire behavior of the parsing process:

- `handle()`: Will invoke the particular `doAction()` method of the current `state`.
- `addPerson`: Once we reach the end state, we need to add a `person` to the list of `persons` we have parsed.
- `parseLine()`: Parse a single line
- `parseLines()`: Parse multiple lines (an array of lines)
- `as.df()`: Extract the information from `persons` list into a data frame object.

Let's go on now with the corresponding implementations:

`handle()` method, delegates on `doAction()` method from the current `state` of the `context`:

```
setMethod(f = "handle", signature = "Context",
    definition = function(obj, input) {
        obj <- doAction(obj@state, input, obj)
        return(obj)
    }
)

setMethod(f = "addPerson", signature = "Context",
  definition = function(obj, person) {
      obj@persons <- c(obj@persons, person)
      return(obj)
  }
)
```

First, we split the original line in an array using the delimiter to identify each element via the R-function `strsplit()`, then iterate for each element as an input value for a given state. The `handle()` method returns again the `context` with the updated information (`state`, `person`, `persons` attribute).

```
setMethod(f = "parseLine", signature = "Context",
  definition = function(obj, s) {
      elements <- strsplit(s, ";")[[1]]
```

```
      # Adding an empty field for considering the end state.
      elements <- c(elements, "")
      n <- length(elements)
      input <- NULL
      for (i in (1:n)) {
        input <- elements[i]
        obj <- handle(obj, input)
      }
      return(obj@person)
  }
)
```

Becuase R makes a copy of the input argument, we need to return the context (`obj`):

```
setMethod(f = "parseLines", signature = "Context",
  definition = function(obj, s) {
      n <- length(s)
      listOfPersons <- list()
      for (i in (1:n)) {
          ipersons <- parseLine(obj, s[i])
          listOfPersons[[i]] <- ipersons
      }
      obj@persons <- listOfPersons
      return(obj)
  }
)
```

The attribute `persons` is a list of instance of S4 `Person` class. This something cannot be coerced to any standard type because R does not know of to treat an instance of a user defined class. The solution is to convert a `Person` into a list, using the `as.list` method previously defined. Then we can apply this function to each element of the list `persons`, via the `lapply()` function. Then in the next invocation to `lappy()` function, now applies the `data.frame` function for converting each element of the `persons.list` into a data frame. Finally, the `rbind()` function is called for adding each element converted as a new row of the data frame generated (for more detail about this see this post)

```
# Sugestion taken from this post:
# http://stackoverflow.com/questions/4227223/r-list-to-data-frame
setMethod(f = "as.df", signature = "Context",
  definition = function(obj) {
    persons <- obj@persons
    persons.list <- lapply(persons, as.list)
    persons.ds <- do.call(rbind, lapply(persons.list, data.frame, stringsAsFactors = FALSE))
    return(persons.ds)
  }
)
```

PUTTING ALL TOGETHER

Finally, lets to test the entire solution. Define the lines to parse where for the second line the address information is missing.

```
s <- c(
    "GREGORY BROWN; 25 NE 25TH; +1-786-987-6543",
    "DAVID SMITH;786-123-4567",
     "ALAN PEREZ; 25 SE 50TH; +1-786-987-5553"
```


www.dbooks.org

```
)
```

Now we initialize the `context`, and parse the lines:

```
context <- new("Context")
context <- parseLines(context, s)
```

Finally obtain the corresponding dataset and print it:

```
df <- as.df(context)
> df
           name    address           phone
1 GREGORY BROWN 25 NE 25TH +1-786-987-6543
2   DAVID SMITH       <NA>    786-123-4567
3    ALAN PEREZ 25 SE 50TH +1-786-987-5553
```

Let's test now the `show` methods:

```
> show(context@persons[[1]])
Person@[name='GREGORY BROWN', address='25 NE 25TH', phone='+1-786-987-6543']
```

And for some sub-state:

```
>show(new("PhoneState"))
PhoneState@[name='phone', pattern='^\s*(\+1(-|\s+))*[0-9]{3}(-|\s+)[0-9]{3}(-|\s+)[0-9]{4}$']
```

Finally, test the `as.list()` method:

```
> as.list(context@persons[[1]])
$name
[1] "GREGORY BROWN"

$address
[1] "25 NE 25TH"

$phone
[1] "+1-786-987-6543"

>
```

CONCLUSION

This example shows how to implement the State pattern, using one of the available mechanisms from R for using the OO paradigm. Nevertheless, the R OO solution is not user-friendly and differs so much from other OOP languages. You need to switch your mindset because the syntax is completely different, it reminds more the functional programming paradigm. For example instead of: `object.setID("A1")` as in Java/C#, for R you have to invoke the method in this way: `setID(object, "A1")`. Therefore you always have to include the object as an input argument to provide the context of the function. On the same way, there is no special `this` class attribute and either a `"."` notation for accessing methods or attributes of the given class. It is more error prompt because to refer a class or methods is done via attribute value (`"Person"`, `"isState"`, etc.).

Said the above, S4 class solution, requires much more lines of codes than a traditional Java/C# languages for doing simple tasks. Anyway, the State Pattern is a good and generic solution for such kind of problems. It simplifies the process delegating the logic into a particular state. Instead of having a big `if-else` block for controlling all situations, we have smaller `if-else` blocks inside on each `State` sub-class implementation for implementing the action to carry out in each state.

Attachment: Here you can download the entire script.

Any suggestion is welcome.

Read Implement State Machine Pattern using S4 Class online:
https://riptutorial.com/r/topic/9126/implement-state-machine-pattern-using-s4-class

www.dbooks.org

Chapter 55: Input and output

Remarks

To construct file paths, for reading or writing, use `file.path`.

Use `dir` to see what files are in a directory.

Examples

Reading and writing data frames

Data frames are R's tabular data structure. They can be written to or read from in a variety of ways.

This example illustrates a couple common situations. See the links at the end for other resources.

Writing

Before making the example data below, make sure you're in a folder you want to write to. Run `getwd()` to verify the folder you're in and read `?setwd` if you need to change folders.

```
set.seed(1)
for (i in 1:3)
  write.table(
    data.frame(id = 1:2, v = sample(letters, 2)),
    file = sprintf("file201%s.csv", i)
  )
```

Now, we have three similarly-formatted CSV files on disk.

Reading

We have three similarly-formatted files (from the last section) to read in. Since these files are related, we should store them together after reading in, in a `list`:

```
file_names = c("file2011.csv", "file2012.csv", "file2013.csv")
file_contents = lapply(setNames(file_names, file_names), read.table)

# $file2011.csv
#   id v
# 1  1 g
# 2  2 j
#
# $file2012.csv
#   id v
```

```
# 1  1 o
# 2  2 w
#
# $file2013.csv
#   id v
# 1  1 f
# 2  2 w
```

To work with this list of files, first examine the structure with `str(file_contents)`, then read about stacking the list with `?rbind` or iterating over the list with `?lapply`.

Further resources

Check out `?read.table` and `?write.table` to extend this example. Also:

- R binary formats (for tables and other objects)
- Plain-text table formats
  - comma-delimited CSVs
  - tab-delimited TSVs
  - Fixed-width formats
- Language-agnostic binary table formats
  - Feather
- Foreign table and spreadsheet formats
  - SAS
  - SPSS
  - Stata
  - Excel
- Relational database table formats
  - MySQL
  - SQLite
  - PostgreSQL

Read Input and output online: https://riptutorial.com/r/topic/5543/input-and-output

www.dbooks.org

Chapter 56: Inspecting packages

Introduction

Packages build on base R. This document explains how to inspect installed packages and their functionality. Related Docs: Installing packages

Remarks

The Comprehensive R Archive Network (CRAN) is the primary package repository.

Examples

View package information

To retrieve information about dplyr package and its functions' descriptions:

```
help(package = "dplyr")
```

No need to load the package first.

View package's built-in data sets

To see built-in data sets from package dplyr

```
 data(package = "dplyr")
```

No need to load the package first.

List a package's exported functions

To get the list of functions within package dplyr, we first must load the package:

```
library(dplyr)
ls("package:dplyr")
```

View Package Version

Conditions: package should be at least installed. If not loaded in the current session, not a problem.

```
  ## Checking package version which was installed at past or
  ## installed currently but not loaded in the current session

  packageVersion("seqinr")
```

```
  # [1] '3.3.3'
  packageVersion("RWeka")
  # [1] '0.4.29'
```

View Loaded packages in Current Session

To check the list of loaded packages

```
search()
```

OR

```
(.packages())
```

Read Inspecting packages online: https://riptutorial.com/r/topic/7408/inspecting-packages

www.dbooks.org

Chapter 57: Installing packages

Syntax

- install.packages(pkgs, lib, repos, method, destdir, dependencies, ...)

Parameters

Parameter	Details
pkgs	character vector of the names of packages. If `repos = NULL`, a character vector of file paths.
lib	character vector giving the library directories where to install the packages.
repos	character vector, the base URL(s) of the repositories to use, can be `NULL` to install from local files
method	download method
destdir	directory where downloaded packages are stored
dependencies	logical indicating whether to also install uninstalled packages which these packages depend on/link to/import/suggest (and so on recursively). Not used if `repos = NULL`.
...	Arguments to be passed to 'download.file' or to the functions for binary installs on OS X and Windows.

Remarks

Related Docs

- Inspecting packages

Examples

Download and install packages from repositories

Packages are collections of R functions, data, and compiled code in a well-defined format. Public (and private) repositories are used to host collections of R packages. The largest collection of R packages is available from CRAN.

Using CRAN

A package can be installed from CRAN using following code:

```
install.packages("dplyr")
```

Where `"dplyr"` is referred to as a character vector.

More than one packages can be installed in one go by using the combine function `c()` and passing a series of character vector of package names:

```
install.packages(c("dplyr", "tidyr", "ggplot2"))
```

In some cases, `install.packages` may prompt for a CRAN mirror or fail, depending on the value of `getOption("repos")`. To prevent this, specify a CRAN mirror as `repos` argument:

```
install.packages("dplyr", repos = "https://cloud.r-project.org/")
```

Using the `repos` argument it is also possible to install from other repositories. For complete information about all the available options, run `?install.packages`.

Most packages require functions, which were implemented in other packages (e.g. the package `data.table`). In order to install a package (or multiple packages) with all the packages, which are used by this given package, the argument `dependencies` should be set to `TRUE`):

```
install.packages("data.table", dependencies = TRUE)
```

Using Bioconductor

Bioconductor hosts a substantial collection of packages related to Bioinformatics. They provide their own package management centred around the `biocLite` function:

```
    ## Try http:// if https:// URLs are not supported
    source("https://bioconductor.org/biocLite.R")
    biocLite()
```

By default this installs a subset of packages that provide the most commonly used functionality. Specific packages can be installed by passing a vector of package names. For example, to install `RImmPort` from Bioconductor:

```
    source("https://bioconductor.org/biocLite.R")
    biocLite("RImmPort")
```

Install package from local source

www.dbooks.org

To install package from local source file:

```
install.packages(path_to_source, repos = NULL, type="source")

install.packages("~/Downloads/dplyr-master.zip", repos=NULL, type="source")
```

Here, `path_to_source` is absolute path of local source file.

Another command that opens a window to choose downloaded zip or tar.gz source files is:

```
install.packages(file.choose(), repos=NULL)
```

Another possible way is using the *GUI based RStudio*:

Step 1: Go to **Tools**.

Step 2: Go to **Install Packages**.

Step 3: In the *Install From* set it as **Package Archive File (.zip; .tar.gz)**

Step 4: Then *Browse* find your package file (say crayon_1.3.1.zip) and *after some time (after it shows the **Package path and file name** in the Package Archive tab)*

Another way to install R package from local source is using `install_local()` function from devtools package.

```
library(devtools)
install_local("~/Downloads/dplyr-master.zip")
```

Install packages from GitHub

To install packages directly from GitHub use the `devtools` package:

```
library(devtools)
install_github("authorName/repositoryName")
```

To install `ggplot2` from github:

```
devtools::install_github("tidyverse/ggplot2")
```

The above command will install the version of `ggplot2` that corresponds to the *master* branch. To install from a different branch of a repository use the `ref` argument to provide the name of the branch. For example, the following command will install the `dev_general` branch of the `googleway` package.

```
devtools::install_github("SymbolixAU/googleway", ref = "dev_general")
```

Another option is to use the `ghit` package. It provides a lightweight alternative for installing packages from github:

```
install.packages("ghit")
ghit::install_github("google/CausalImpact")
```

To install a package that is in a **private** repository on Github, generate a **personal access token** at http://www.github.com/settings/tokens/ (See ?install_github for documentation on the same). Follow these steps:

1. ```
   install.packages(c("curl", "httr"))
   ```

2. ```
   config = httr::config(ssl_verifypeer = FALSE)
   ```

3. ```
    install.packages("RCurl")
    options(RCurlOptions = c(getOption("RCurlOptions"),ssl.verifypeer = FALSE,
   ssl.verifyhost = FALSE ) )
   ```

4. ```
   getOption("RCurlOptions")
   ```

   You should see the following:

   ```
   ssl.verifypeer ssl.verifyhost

   FALSE          FALSE
   ```

5. ```
   library(httr)
   set_config(config(ssl_verifypeer = 0L))
   ```

   This prevents the common error: "Peer certificate cannot be authenticated with given CA certificates"

6. Finally, use the following command to install your package seamlessly

   ```
   install_github("username/package_name",auth_token="abc")
   ```

Alternatively, set an environment variable `GITHUB_PAT`, using

```
Sys.setenv(GITHUB_PAT = "access_token")
devtools::install_github("organisation/package_name")
```

The PAT generated in Github is only visible once, i.e., when created initially, so its prudent to save that token in `.Rprofile`. This is also helpful if the organisation has many private repositories.

Using a CLI package manager -- basic pacman usage

`pacman` is a simple package manager for R.

`pacman`

www.dbooks.org
```

allows a user to compactly load all desired packages, installing any which are missing (and their dependencies), with a single command, `p_load`. `pacman` does not require the user to type quotation marks around a package name. Basic usage is as follows:

```
p_load(data.table, dplyr, ggplot2)
```

The only package requiring a `library`, `require`, or `install.packages` statement with this approach is `pacman` itself:

```
library(pacman)
p_load(data.table, dplyr, ggplot2)
```

or, equally valid:

```
pacman::p_load(data.table, dplyr, ggplot2)
```

In addition to saving time by requiring less code to manage packages, `pacman` also facilitates the construction of reproducible code by installing any needed packages if and only if they are not already installed.

Since you may not be sure if `pacman` is installed in the library of a user who will use your code (or by yourself in future uses of your own code) a best practice is to include a conditional statement to install `pacman` if it is not already loaded:

```
if(!(require(pacman)) install.packages("pacman")
pacman::p_load(data.table, dplyr, ggplot2)
```

Install local development version of a package

While working on the development of an R package it is often necessary to install the latest version of the package. This can be achieved by first building a source distribution of the package (on the command line)

```
R CMD build my_package
```

and then installing it in R. Any running R sessions with previous version of the package loaded will need to reload it.

```
unloadNamespace("my_package")
library(my_package)
```

A more convenient approach uses the `devtools` package to simplify the process. In an R session with the working directory set to the package directory

```
devtools::install()
```

will build, install and reload the package.

Read Installing packages online: https://riptutorial.com/r/topic/1719/installing-packages

www.dbooks.org

Chapter 58: Introduction to Geographical Maps

Introduction

See also I/O for geographic data

Examples

Basic map-making with map() from the package maps

The function `map()` from the package `maps` provides a simple starting point for creating maps with R.

A basic world map can be drawn as follows:

```
require(maps)
map()
```



The color of the outline can be changed by setting the color parameter, `col`, to either the character

name or hex value of a color:

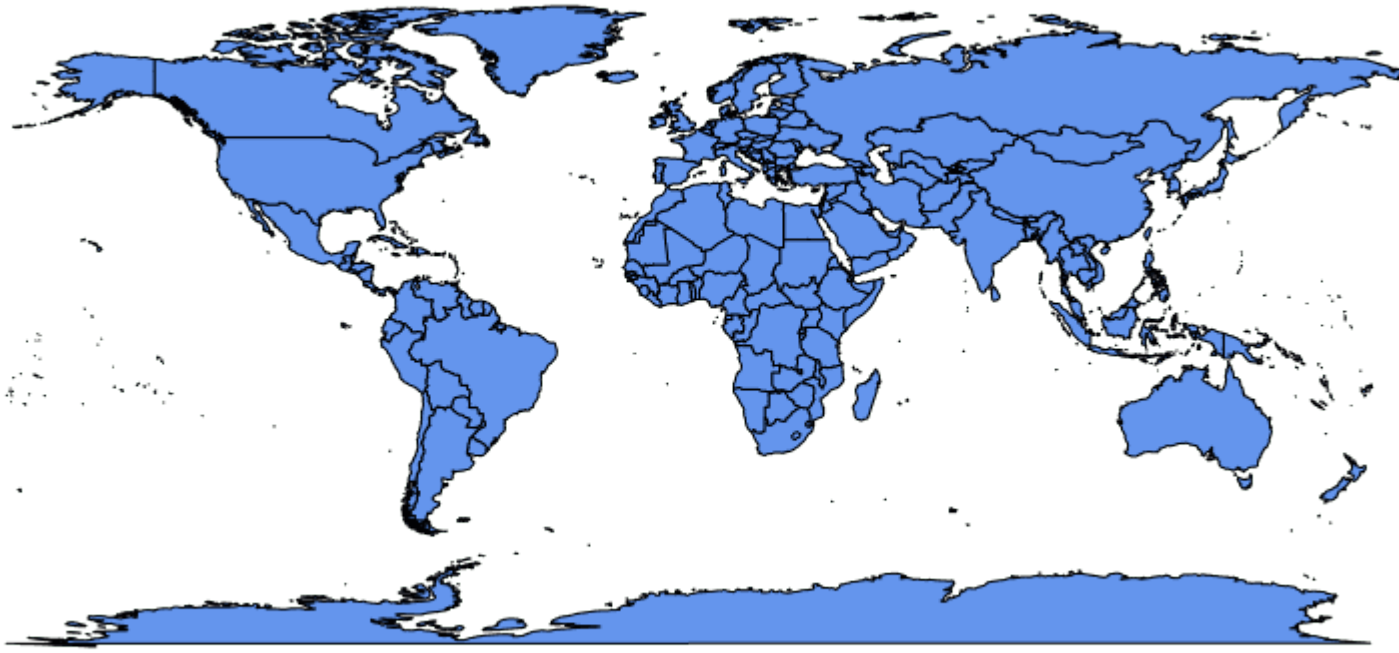
```
require(maps)
map(col = "cornflowerblue")
```



To fill land masses with the color in `col` we can set `fill = TRUE`:

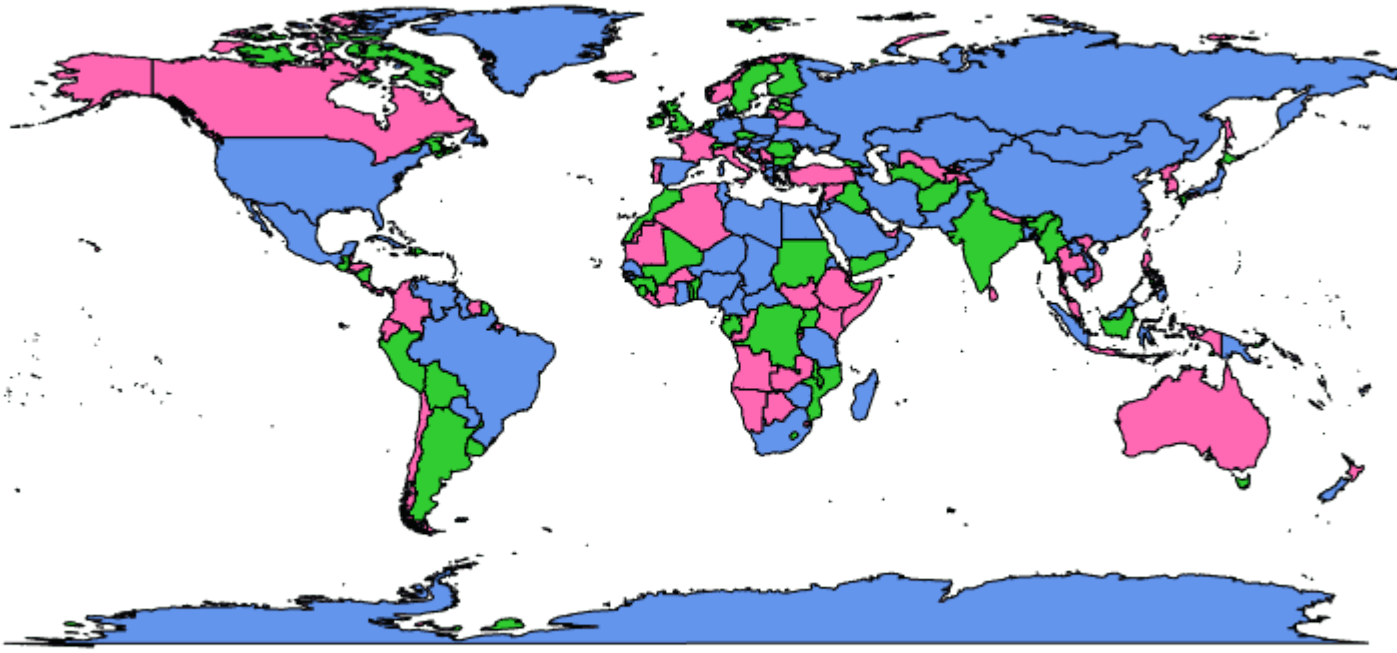
```
require(maps)
map(fill = TRUE, col = c("cornflowerblue"))
```


www.dbooks.org



A vector of any length may be supplied to `col` when `fill = TRUE` is also set:

```
require(maps)
map(fill = TRUE, col = c("cornflowerblue", "limegreen", "hotpink"))
```



In the example above colors from `col` are assigned arbitrarily to polygons in the map representing regions and colors are recycled if there are fewer colors than polygons.

We can also use color coding to represent a statistical variable, which may optionally be described in a legend. A map created as such is known as a "choropleth".

The following choropleth example sets the first argument of `map()`, which is `database` to `"county"` and `"state"` to color code unemployment using data from the built-in datasets `unemp` and `county.fips` while overlaying state lines in white:

```
require(maps)
if(require(mapproj)) {    # mapproj is used for  projection="polyconic"
  # color US county map by 2009 unemployment rate
  # match counties to map using FIPS county codes
  # Based on J's solution to the "Choropleth Challenge"
  # Code improvements by Hack-R (hack-r.github.io)

  # load data
  # unemp includes data for some counties not on the "lower 48 states" county
  # map, such as those in Alaska, Hawaii, Puerto Rico, and some tiny Virginia
  #  cities
  data(unemp)
  data(county.fips)

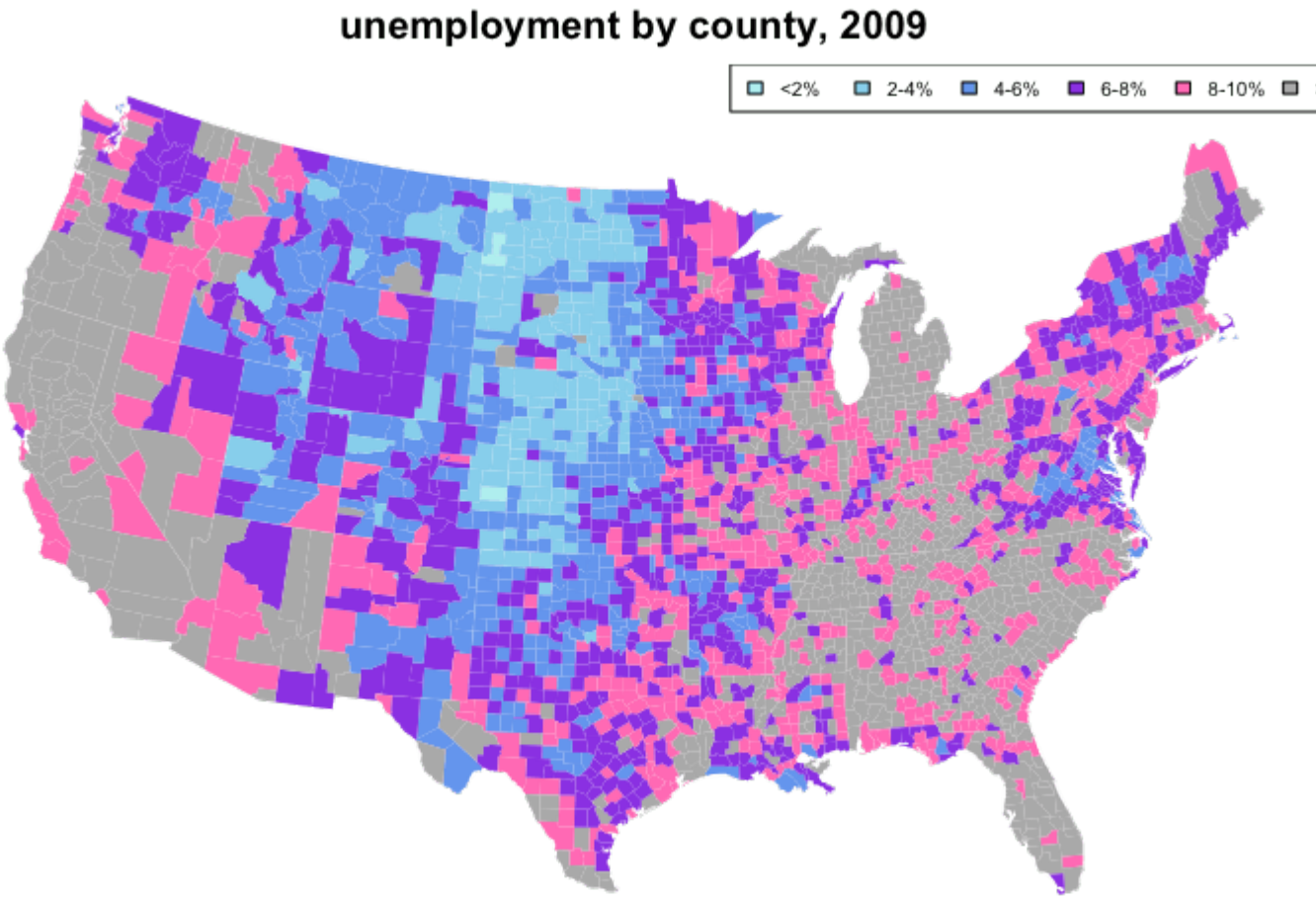
  # define color buckets
  colors = c("paleturquoise", "skyblue", "cornflowerblue", "blueviolet", "hotpink",
"darkgrey")
```


www.dbooks.org

```
  unemp$colorBuckets <- as.numeric(cut(unemp$unemp, c(0, 2, 4, 6, 8, 10, 100)))
  leg.txt <- c("<2%", "2-4%", "4-6%", "6-8%", "8-10%", ">10%")

  # align data with map definitions by (partial) matching state,county
  # names, which include multiple polygons for some counties
  cnty.fips <- county.fips$fips[match(map("county", plot=FALSE)$names,
                                      county.fips$polyname)]
  colorsmatched <- unemp$colorBuckets[match(cnty.fips, unemp$fips)]

  # draw map
  par(mar=c(1, 1, 2, 1) + 0.1)
  map("county", col = colors[colorsmatched], fill = TRUE, resolution = 0,
      lty = 0, projection = "polyconic")
  map("state", col = "white", fill = FALSE, add = TRUE, lty = 1, lwd = 0.1,
      projection="polyconic")
  title("unemployment by county, 2009")
  legend("topright", leg.txt, horiz = TRUE, fill = colors, cex=0.6)
}
```



50 State Maps and Advanced Choropleths with Google Viz

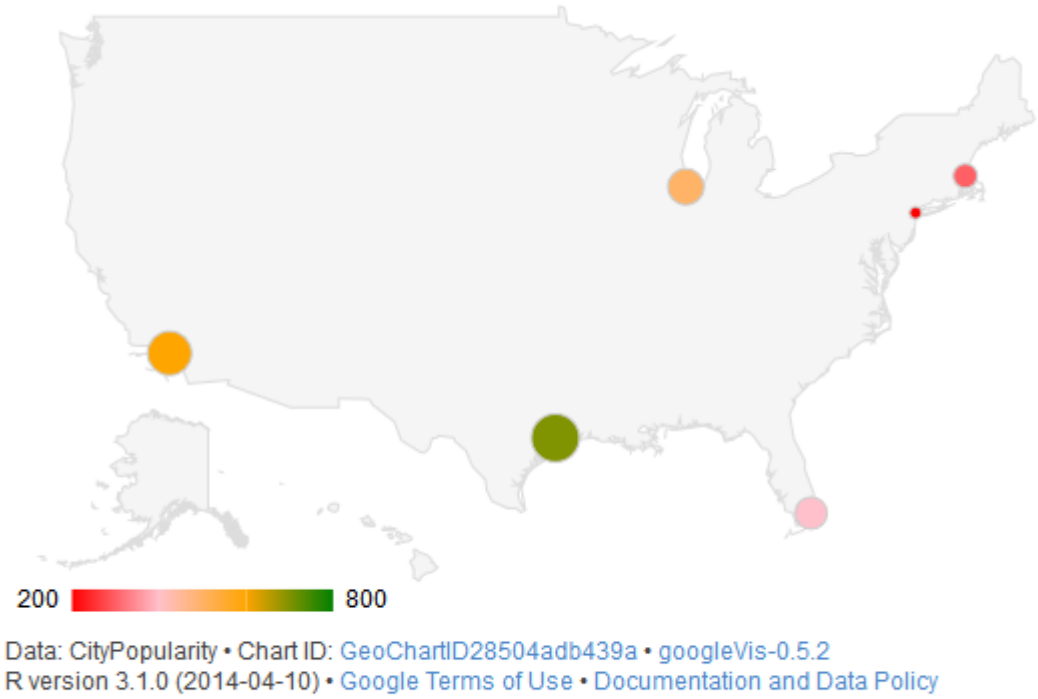
A common question is how to juxtapose (combine) physically separate geographical regions on the same map, such as in the case of a choropleth describing all 50 American states (The mainland with Alaska and Hawaii juxtaposed).

Creating an attractive 50 state map is simple when leveraging Google Maps. Interfaces to

Google's API include the packages `googleVis`, `ggmap`, and `RgoogleMaps`.

```
require(googleVis)

G4 <- gvisGeoChart(CityPopularity, locationvar='City', colorvar='Popularity',
                   options=list(region='US', height=350,
                                displayMode='markers',
                                colorAxis="{values:[200,400,600,800],
                                 colors:[\'red', \'pink\', \'orange',\'green']}")
)
plot(G4)
```



The function `gvisGeoChart()` requires far less coding to create a choropleth compared to older mapping methods, such as `map()` from the package `maps`. The `colorvar` parameter allows easy coloring of a statistical variable, at a level specified by the `locationvar` parameter. The various options passed to `options` as a list allow customization of the map's details such as size (`height`), shape (`markers`), and color coding (`colorAxis` and `colors`).

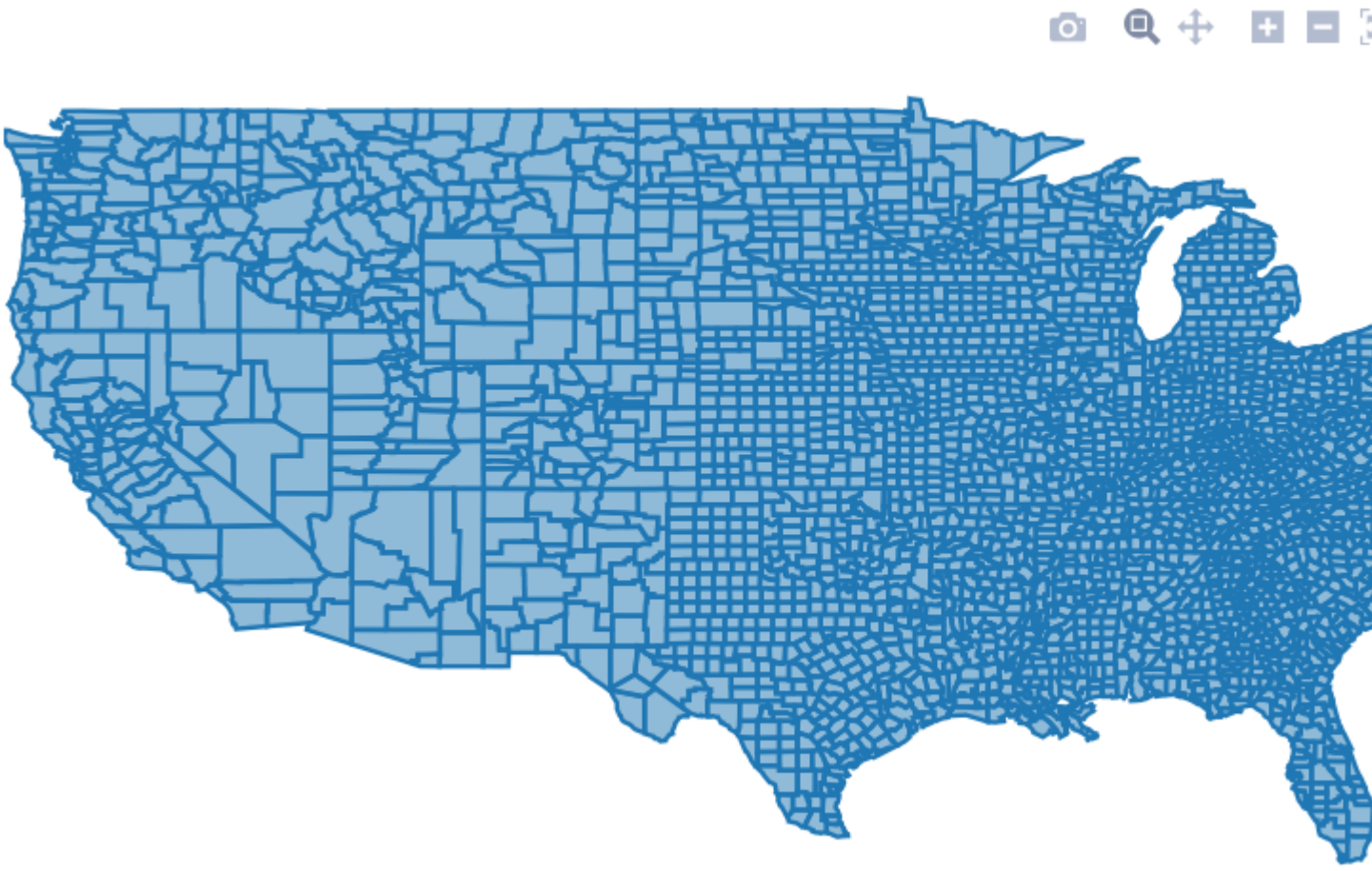
Interactive plotly maps

The `plotly` package allows many kind of interactive plots, including maps. There are a few ways to create a map in `plotly`. Either supply the map data yourself (via `plot_ly()` or `ggplotly()`), use plotly's "native" mapping capabilities (via `plot_geo()` or `plot_mapbox()`), or even a combination of both. An example of supplying the map yourself would be:

```
library(plotly)
map_data("county") %>%
    group_by(group) %>%
    plot_ly(x = ~long, y = ~lat) %>%
```


www.dbooks.org

```
    add_polygons() %>%
    layout(
        xaxis = list(title = "", showgrid = FALSE, showticklabels = FALSE),
        yaxis = list(title = "", showgrid = FALSE, showticklabels = FALSE)
    )
```



For a combination of both approaches, swap `plot_ly()` for `plot_geo()` or `plot_mapbox()` in the above example. See the plotly book for more examples.

The next example is a "strictly native" approach that leverages the layout.geo attribute to set the aesthetics and zoom level of the map. It also uses the database `world.cities` from `maps` to filter the Brazilian cities and plot them on top of the "native" map.

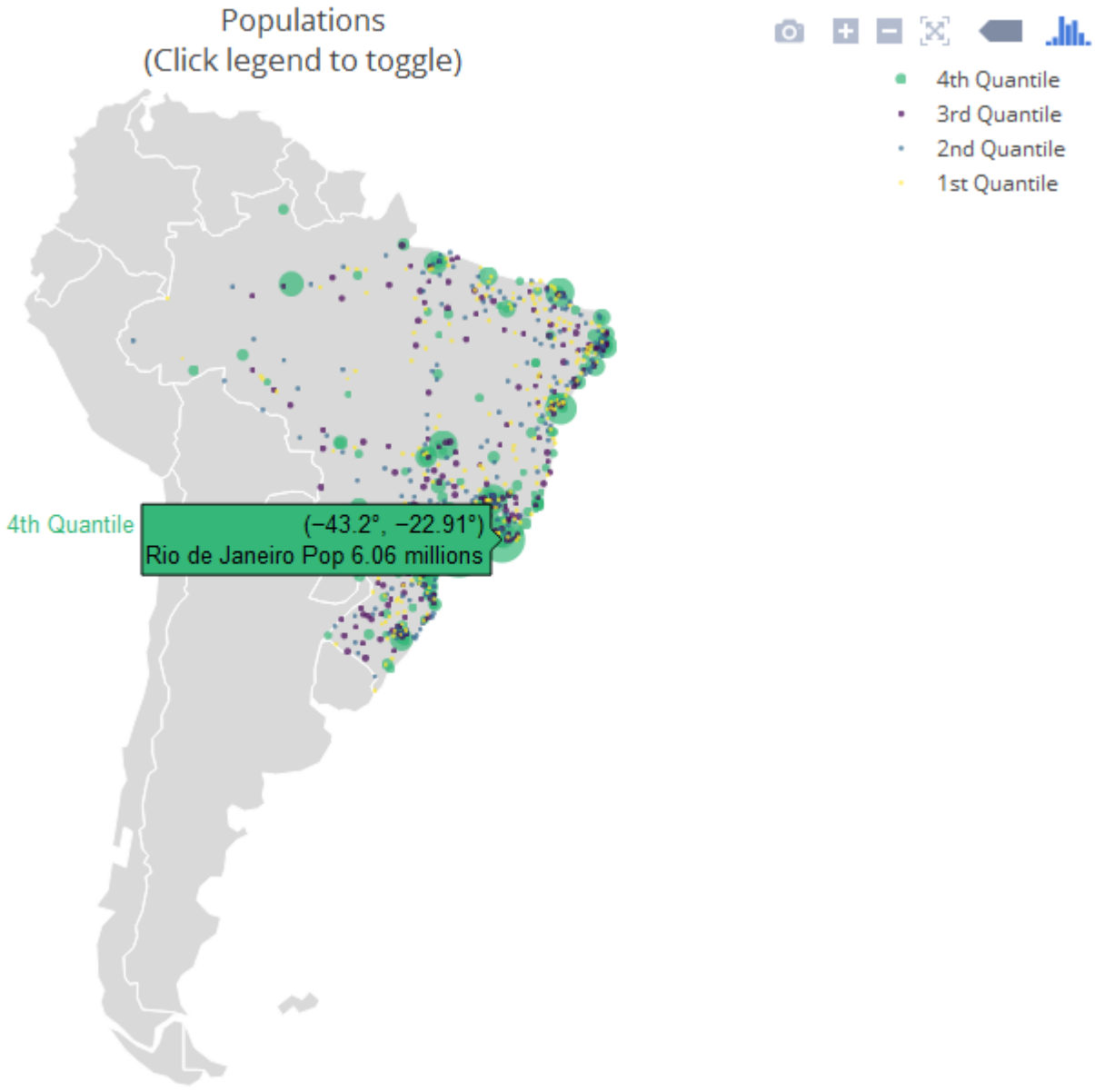
The main variables: `poph`is a text with the city and its population (which is shown upon mouse hover); `q`is a ordered factor from the population's quantile. `ge` has information for the layout of the maps. See the package documentation for more information.

```
library(maps)
dfb <- world.cities[world.cities$country.etc=="Brazil",]
library(plotly)
dfb$poph <- paste(dfb$name, "Pop", round(dfb$pop/1e6,2), " millions")
dfb$q <- with(dfb, cut(pop, quantile(pop), include.lowest = T))
levels(dfb$q) <- paste(c("1st", "2nd", "3rd", "4th"), "Quantile")
dfb$q <- as.ordered(dfb$q)

ge <- list(
```

```
  scope = 'south america',
  showland = TRUE,
  landcolor = toRGB("gray85"),
  subunitwidth = 1,
  countrywidth = 1,
  subunitcolor = toRGB("white"),
  countrycolor = toRGB("white")
)

plot_geo(dfb, lon = ~long, lat = ~lat, text = ~poph,
    marker = ~list(size = sqrt(pop/10000) + 1, line = list(width = 0)),
    color = ~q, locationmode = 'country names') %>%
layout(geo = ge, title = 'Populations<br>(Click legend to toggle)')
```



Making Dynamic HTML Maps with Leaflet

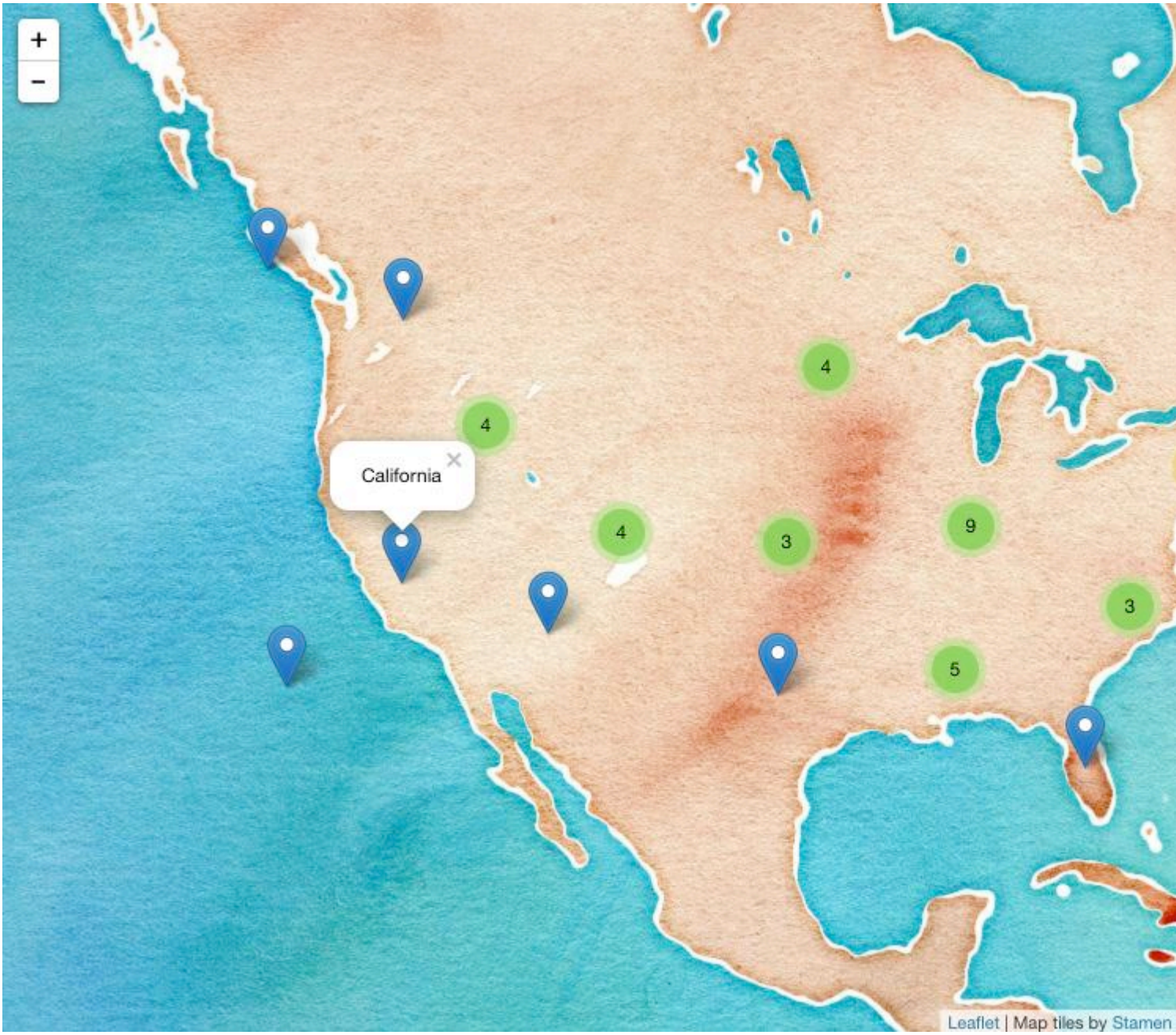
Leaflet is an open-source JavaScript library for making dynamic maps for the web. RStudio wrote R bindings for Leaflet, available through its `leaflet` package, built with `htmlwidgets`. Leaflet maps integrate well with the RMarkdown and Shiny ecosystems.

The interface is piped, using a `leaflet()` function to initialize a map and subsequent functions adding (or removing) map layers. Many kinds of layers are available, from markers with popups to polygons for creating choropleth maps. Variables in the data.frame passed to `leaflet()` are accessed via function-style `~` quotation.

To map the `state.name` and `state.center` datasets:

```
library(leaflet)

data.frame(state.name, state.center) %>%
    leaflet() %>%
    addProviderTiles('Stamen.Watercolor') %>%
    addMarkers(lng = ~x, lat = ~y,
               popup = ~state.name,
               clusterOptions = markerClusterOptions())
```



(Screenshot; click for dynamic version.)

Dynamic Leaflet maps in Shiny applications

The Leaflet package is designed to integerate with Shiny

In the **ui** you call `leafletOutput()` and in the server you call `renderLeaflet()`

```
library(shiny)
library(leaflet)

ui <- fluidPage(
    leafletOutput("my_leaf")
)

server <- function(input, output, session){

    output$my_leaf <- renderLeaflet({

        leaflet() %>%
            addProviderTiles('Hydda.Full') %>%
            setView(lat = -37.8, lng = 144.8, zoom = 10)

    })

}

shinyApp(ui, server)
```

However, reactive inputs that affect the `renderLeaflet` expression will cause the entire map to be redrawn each time the reactive element is updated.

Therefore, to modify a map that's already running you should use the `leafletProxy()` function.

Normally you use `leaflet` to create the static aspects of the map, and `leafletProxy` to manage the dynamic elements, for example:

```
library(shiny)
library(leaflet)

ui <- fluidPage(
    sliderInput(inputId = "slider",
                label = "values",
                min = 0,
                max = 100,
                value = 0,
                step = 1),
    leafletOutput("my_leaf")
)

server <- function(input, output, session){
    set.seed(123456)
    df <- data.frame(latitude = sample(seq(-38.5, -37.5, by = 0.01), 100),
                     longitude = sample(seq(144.0, 145.0, by = 0.01), 100),
                     value = seq(1,100))

    ## create static element
    output$my_leaf <- renderLeaflet({
```


www.dbooks.org

```
        leaflet() %>%
            addProviderTiles('Hydda.Full') %>%
            setView(lat = -37.8, lng = 144.8, zoom = 8)

    })

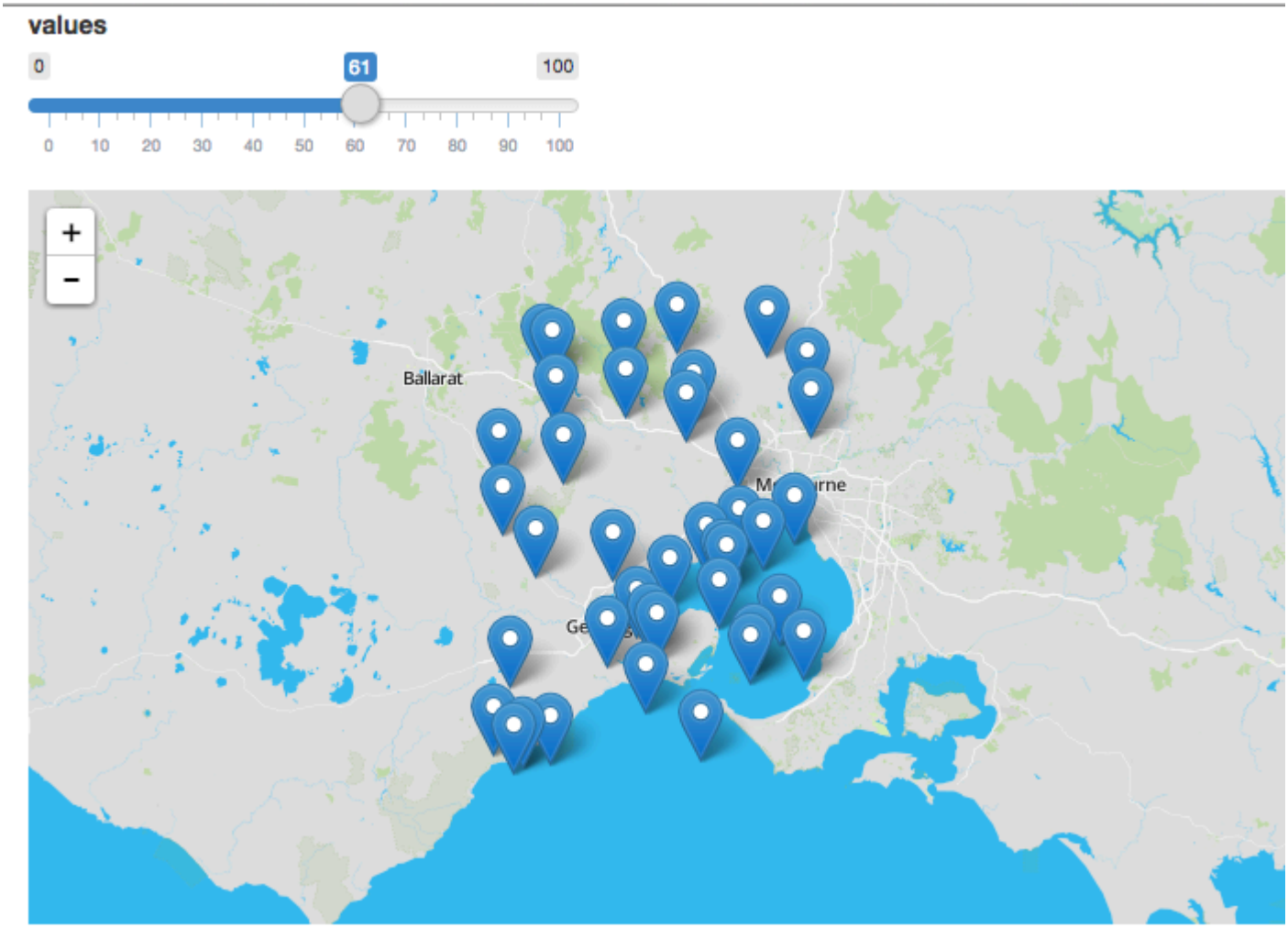
    ## filter data
    df_filtered <- reactive({
        df[df$value >= input$slider, ]
    })

    ## respond to the filtered data
    observe({

        leafletProxy(mapId = "my_leaf", data = df_filtered()) %>%
            clearMarkers() %>%   ## clear previous markers
            addMarkers()
    })

}

shinyApp(ui, server)
```



Read Introduction to Geographical Maps online: https://riptutorial.com/r/topic/1372/introduction-to-geographical-maps

Chapter 59: Introspection

Examples

Functions for Learning about Variables

Often in `R` you'll want to know things about an object or variable you're working with. This can be useful when reading someone else's code or even your own, especially when using packages that are new to you.

Suppose we create a variable `a`:

```
a <- matrix(1:9, 3, 3)
```

What data type is this? You can find out with

```
> class(a)
[1] "matrix"
```

It's a matrix, so matrix operations will work on it:

```
> a %*% t(a)
      [,1] [,2] [,3]
[1,]   66   78   90
[2,]   78   93  108
[3,]   90  108  126
```

What are the dimensions of `a`?

```
> dim(a)
[1] 3 3
> nrow(a)
[1] 3
> ncol(a)
[2] 3
```

Other useful functions that work for different data types are `head`, `tail`, and `str`:

```
> head(a, 1)
     [,1] [,2] [,3]
[1,]    1    4    7
> tail(a, 1)
     [,1] [,2] [,3]
[3,]    3    6    9
> str(a)
int [1:3, 1:3] 1 2 3 4 5 6 7 8 9
```

These are much more useful for large objects (such as big datasets). `str` is also great for learning about the nesting of lists. Now reshape `a` like so:

www.dbooks.org

```
a <- c(a)
```

Does the class remain the same?

```
> class(a)
[1] "integer"
```

No, `a` is not a matrix anymore. I won't get a good answer if I ask for dimensions now:

```
> dim(a)
NULL
```

Instead, I can ask for the length:

```
> length(a)
[1] 9
```

What about now:

```
> class(a * 1.0)
[1] "numeric"
```

Often you may work with `data.frames`:

```
a <- as.data.frame(a)
names(a) <- c("var1", "var2", "var3")
```

See the variable names:

```
> names(a)
[1] "var1" "var2" "var3"
```

These functions can help many ways when using `R`.

Read Introspection online: https://riptutorial.com/r/topic/3565/introspection

Chapter 60: JSON

Examples

JSON to / from R objects

The `jsonlite` package is a fast JSON parser and generator optimized for statistical data and the web. The two main functions used to read and write JSON are `fromJSON()` and `toJSON()` respecitively, and are designed to work with `vectors`, `matrices` and `data.frames`, and streams of JSON from the web.

Create a JSON array from a vector, and vice versa

```
library(jsonlite)

## vector to JSON
toJSON(c(1,2,3))
# [1,2,3]

fromJSON('[1,2,3]')
# [1] 1 2 3
```

Create a named JSON array from a list, and vice versa

```
toJSON(list(myVec = c(1,2,3)))
# {"myVec":[1,2,3]}

fromJSON('{"myVec":[1,2,3]}')
# $myVec
# [1] 1 2 3
```

More complex list structures

```
## list structures
lst <- list(a = c(1,2,3),
            b = list(letters[1:6]))

toJSON(lst)
# {"a":[1,2,3],"b":[["a","b","c","d","e","f"]]}


fromJSON('{"a":[1,2,3],"b":[["a","b","c","d","e","f"]]} ')
# $a
# [1] 1 2 3
#
# $b
# [,1] [,2] [,3] [,4] [,5] [,6]
# [1,] "a"  "b"  "c"  "d"  "e"  "f"
```

Create JSON from a `data.frame`, and vice versa

www.dbooks.org

```
## converting a data.frame to JSON
df <- data.frame(id = seq_along(1:10),
                 val = letters[1:10])

toJSON(df)
#
[{"id":1,"val":"a"},{"id":2,"val":"b"},{"id":3,"val":"c"},{"id":4,"val":"d"},{"id":5,"val":"e"},{"id":6


## reading a JSON string
fromJSON('[{"id":1,"val":"a"},{"id":2,"val":"b"},{"id":3,"val":"c"},{"id":4,"val":"d"},{"id":5,"val":"e

#     id val
# 1   1   a
# 2   2   b
# 3   3   c
# 4   4   d
# 5   5   e
# 6   6   f
# 7   7   g
# 8   8   h
# 9   9   i
# 10 10   j
```

Read JSON direct from the internet

```
## Reading JSON from URL
googleway_issues <- fromJSON("https://api.github.com/repos/SymbolixAU/googleway/issues")

googleway_issues$url
# [1] "https://api.github.com/repos/SymbolixAU/googleway/issues/20"
"https://api.github.com/repos/SymbolixAU/googleway/issues/19"
# [3] "https://api.github.com/repos/SymbolixAU/googleway/issues/14"
"https://api.github.com/repos/SymbolixAU/googleway/issues/11"
# [5] "https://api.github.com/repos/SymbolixAU/googleway/issues/9"
"https://api.github.com/repos/SymbolixAU/googleway/issues/5"
# [7] "https://api.github.com/repos/SymbolixAU/googleway/issues/2"
```

Read JSON online: https://riptutorial.com/r/topic/2460/json

Chapter 61: Linear Models (Regression)

Syntax

- lm(formula, data, subset, weights, na.action, method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE, contrasts = NULL, offset, ...)

Parameters

Parameter	Meaning
formula	a formula in *Wilkinson-Rogers* notation; `response ~ ...` where `...` contains terms corresponding to variables in the environment or in the data frame specified by the `data` argument
data	data frame containing the response and predictor variables
subset	a vector specifying a subset of observations to be used: may be expressed as a logical statement in terms of the variables in `data`
weights	analytical weights (see *Weights* section above)
na.action	how to handle missing (`NA`) values: see `?na.action`
method	how to perform the fitting. Only choices are `"qr"` or `"model.frame"` (the latter returns the model frame without fitting the model, identical to specifying `model=TRUE`)
model	whether to store the model frame in the fitted object
x	whether to store the model matrix in the fitted object
y	whether to store the model response in the fitted object
qr	whether to store the QR decomposition in the fitted object
singular.ok	whether to allow *singular fits*, models with collinear predictors (a subset of the coefficients will automatically be set to `NA` in this case
contrasts	a list of contrasts to be used for particular factors in the model; see the `contrasts.arg` argument of `?model.matrix.default`. Contrasts can also be set with `options()` (see the `contrasts` argument) or by assigning the `contrast` attributes of a factor (see `?contrasts`)
offset	used to specify an *a priori* known component in the model. May also be specified as part of the formula. See `?model.offset`

www.dbooks.org

Parameter	Meaning
...	additional arguments to be passed to lower-level fitting functions (`lm.fit()` or `lm.wfit()`)

Examples

Linear regression on the mtcars dataset

The built-in mtcars data frame contains information about 32 cars, including their weight, fuel efficiency (in miles-per-gallon), speed, etc. (To find out more about the dataset, use `help(mtcars)`).

If we are interested in the relationship between fuel efficiency (`mpg`) and weight (`wt`) we may start plotting those variables with:

```
plot(mpg ~ wt, data = mtcars, col=2)
```

The plots shows a (linear) relationship!. Then if we want to perform linear regression to determine the coefficients of a linear model, we would use the `lm` function:

```
fit <- lm(mpg ~ wt, data = mtcars)
```

The `~` here means "explained by", so the formula `mpg ~ wt` means we are predicting mpg as explained by wt. The most helpful way to view the output is with:

```
summary(fit)
```

Which gives the output:

```
Call:
lm(formula = mpg ~ wt, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.5432 -2.3647 -0.1252  1.4096  6.8727

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
wt           -5.3445     0.5591  -9.559 1.29e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 3.046 on 30 degrees of freedom
Multiple R-squared:  0.7528,    Adjusted R-squared:  0.7446
F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

This provides information about:

- the estimated slope of each coefficient (`wt` and the y-intercept), which suggests the best-fit

prediction of mpg is `37.2851 + (-5.3445) * wt`

- The p-value of each coefficient, which suggests that the intercept and weight are probably not due to chance
- Overall estimates of fit such as R^2 and adjusted R^2, which show how much of the variation in `mpg` is explained by the model

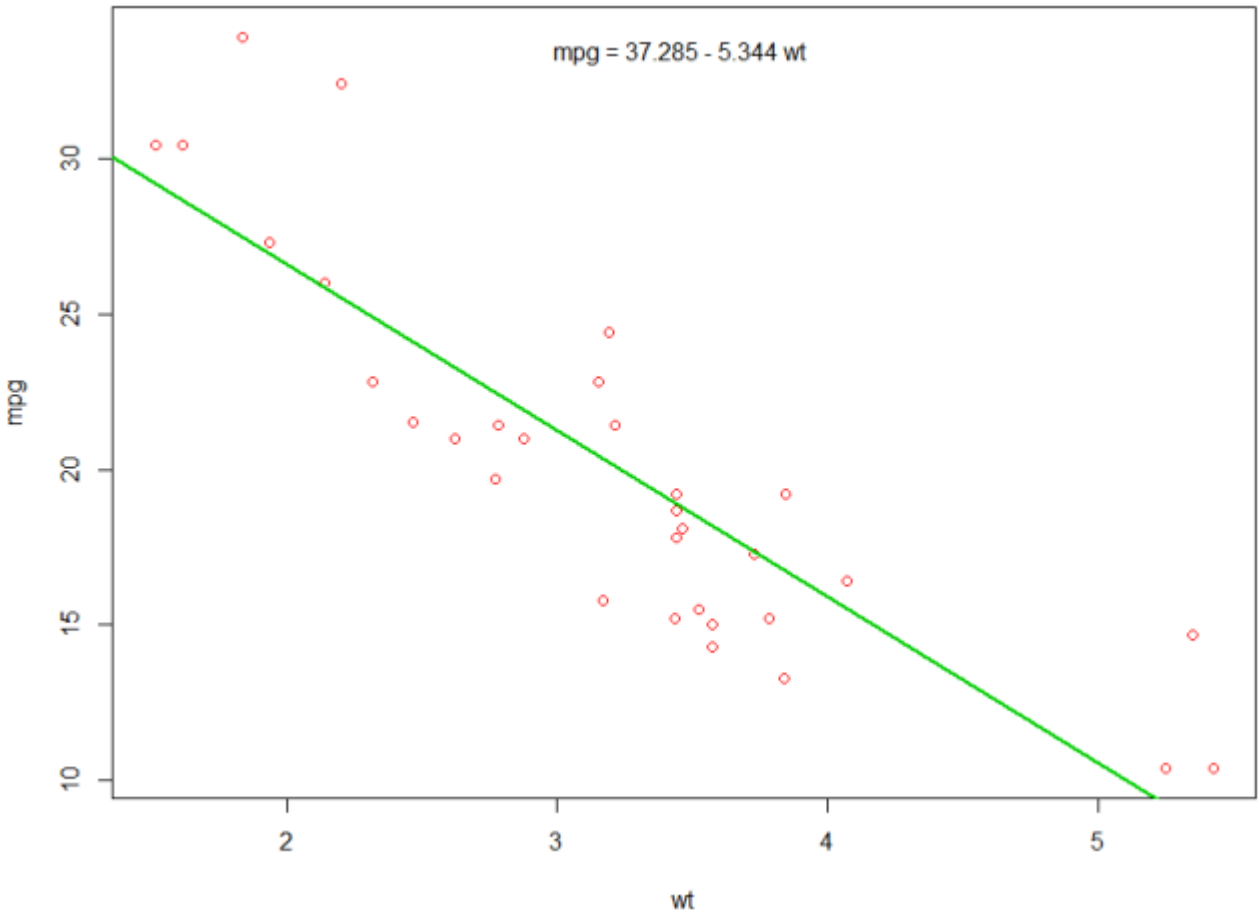
We could add a line to our first plot to show the predicted `mpg`:

```
abline(fit,col=3,lwd=2)
```

It is also possible to add the equation to that plot. First, get the coefficients with `coef`. Then using `paste0` we collapse the coefficients with appropriate variables and `+/-`, to built the equation. Finally, we add it to the plot using `mtext`:

```
bs <- round(coef(fit), 3)
lmlab <- paste0("mpg = ", bs[1],
             ifelse(sign(bs[2])==1, " + ", " - "), abs(bs[2]), " wt ")
mtext(lmlab, 3, line=-2)
```

The result is:



Plotting The Regression (base)

Continuing on the `mtcars` example, here is a simple way to produce a plot of your linear regression that is potentially suitable for publication.

www.dbooks.org

First fit the linear model and

```
fit <- lm(mpg ~ wt, data = mtcars)
```

Then plot the two variables of interest and add the regression line within the definition domain:

```
plot(mtcars$wt,mtcars$mpg,pch=18, xlab = 'wt',ylab = 'mpg')
lines(c(min(mtcars$wt),max(mtcars$wt)),
as.numeric(predict(fit, data.frame(wt=c(min(mtcars$wt),max(mtcars$wt))))))
```

Almost there! The last step is to add to the plot, the regression equation, the rsquare as well as the correlation coefficient. This is done using the `vector` function:

```
rp = vector('expression',3)
rp[1] = substitute(expression(italic(y) == MYOTHERVALUE3 + MYOTHERVALUE4 %*% x),
          list(MYOTHERVALUE3 = format(fit$coefficients[1], digits = 2),
                        MYOTHERVALUE4 = format(fit$coefficients[2], digits = 2)))[2]
rp[2] = substitute(expression(italic(R)^2 == MYVALUE),
             list(MYVALUE = format(summary(fit)$adj.r.squared,dig=3)))[2]
rp[3] = substitute(expression(Pearson-R == MYOTHERVALUE2),
             list(MYOTHERVALUE2 = format(cor(mtcars$wt,mtcars$mpg), digits = 2)))[2]

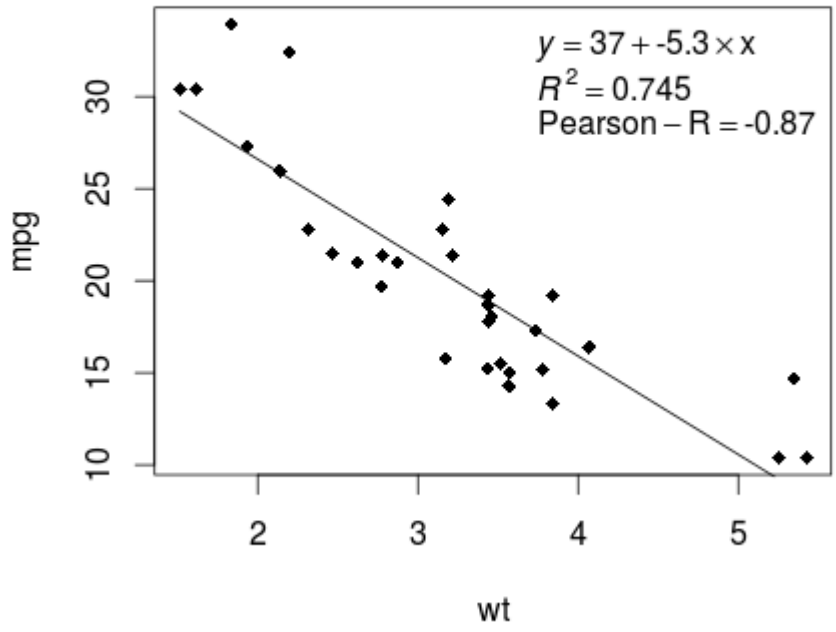
legend("topright", legend = rp, bty = 'n')
```

Note that you can add any other parameter such as the RMSE by adapting the vector function. Imagine you want a legend with 10 elements. The vector definition would be the following:

```
rp = vector('expression',10)
```

and you will need to defined `r[1]`.... to `r[10]`

Here is the output:



Weighting

Sometimes we want the model to give more weight to some data points or examples than others. This is possible by specifying the weight for the input data while learning the model. There are generally two kinds of scenarios where we might use non-uniform weights over the examples:

- Analytic Weights: Reflect the different levels of precision of different observations. For example, if analyzing data where each observation is the average results from a geographic area, the analytic weight is proportional to the inverse of the estimated variance. Useful when dealing with averages in data by providing a proportional weight given the number of observations. Source
- Sampling Weights (Inverse Probability Weights - IPW): a statistical technique for calculating statistics standardized to a population different from that in which the data was collected. Study designs with a disparate sampling population and population of target inference (target population) are common in application. Useful when dealing with data that have missing values. Source

The `lm()` function does analytic weighting. For sampling weights the `survey` package is used to build a survey design object and run `svyglm()`. By default, the `survey` package uses sampling weights. (NOTE: `lm()`, and `svyglm()` with family `gaussian()` will all produce the same point estimates, because they both solve for the coefficients by minimizing the weighted least squares. They differ in how standard errors are calculated.)

Test Data

www.dbooks.org

```
data <- structure(list(lexptot = c(9.1595012302023, 9.86330744180814,
8.92372556833205, 8.58202430280175, 10.1133857229336), progvillm = c(1L,
1L, 1L, 1L, 0L), sexhead = c(1L, 1L, 0L, 1L, 1L), agehead = c(79L,
43L, 52L, 48L, 35L), weight = c(1.04273509979248, 1.01139605045319,
1.01139605045319, 1.01139605045319, 0.76305216550827)), .Names = c("lexptot",
"progvillm", "sexhead", "agehead", "weight"), class = c("tbl_df",
"tbl", "data.frame"), row.names = c(NA, -5L))
```

Analytic Weights

```
lm.analytic <- lm(lexptot ~ progvillm + sexhead + agehead,
                          data = data, weight = weight)
summary(lm.analytic)
```

Output

```
Call:
lm(formula = lexptot ~ progvillm + sexhead + agehead, data = data,
    weights = weight)

Weighted Residuals:
         1          2          3          4          5
 9.249e-02  5.823e-01  0.000e+00 -6.762e-01 -1.527e-16

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 10.016054   1.744293   5.742    0.110
progvillm   -0.781204   1.344974  -0.581    0.665
sexhead      0.306742   1.040625   0.295    0.818
agehead     -0.005983   0.032024  -0.187    0.882

Residual standard error: 0.8971 on 1 degrees of freedom
Multiple R-squared:  0.467, Adjusted R-squared:  -1.132
F-statistic: 0.2921 on 3 and 1 DF,  p-value: 0.8386
```

Sampling Weights (IPW)

```
library(survey)
data$X <- 1:nrow(data)             # Create unique id

# Build survey design object with unique id, ipw, and data.frame
des1 <- svydesign(id = ~X,  weights = ~weight, data = data)

# Run glm with survey design object
prog.lm <- svyglm(lexptot ~ progvillm + sexhead + agehead, design=des1)
```

Output

```
Call:
svyglm(formula = lexptot ~ progvillm + sexhead + agehead, design = des1)

Survey design:
svydesign(id = ~X, weights = ~weight, data = data)
```

```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 10.016054   0.183942  54.452   0.0117 *
progvillm   -0.781204   0.640372  -1.220   0.4371
sexhead      0.306742   0.397089   0.772   0.5813
agehead     -0.005983   0.014747  -0.406   0.7546
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for gaussian family taken to be 0.2078647)

Number of Fisher Scoring iterations: 2
```

Checking for nonlinearity with polynomial regression

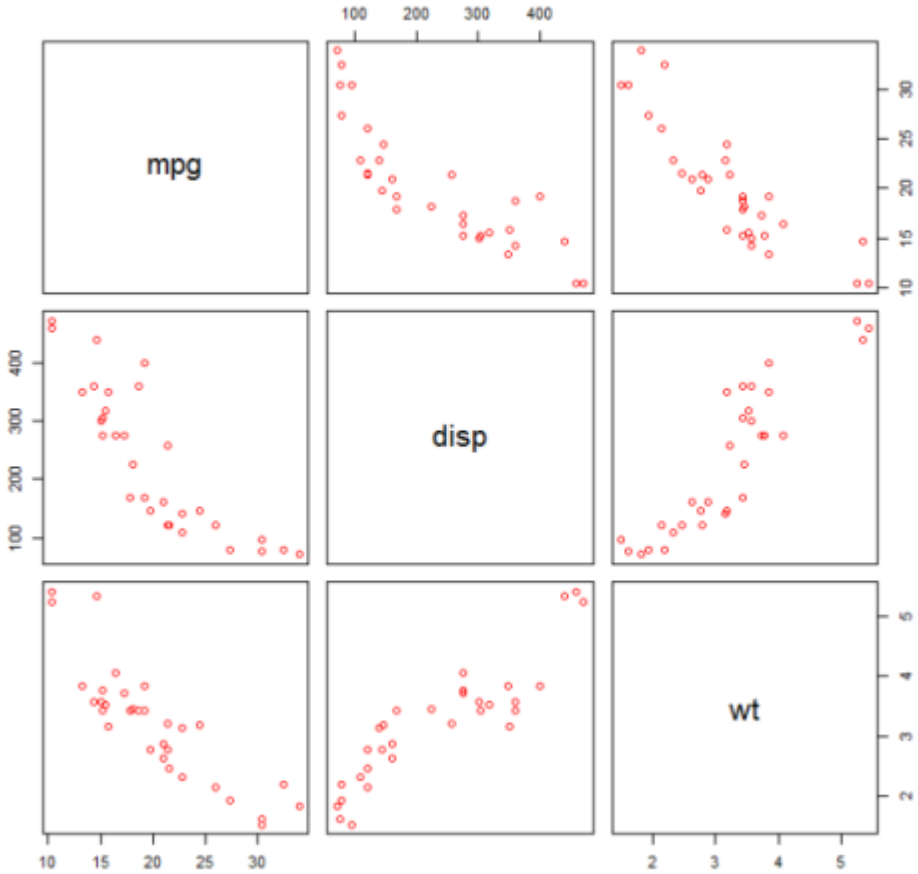
Sometimes when working with linear regression we need to check for non-linearity in the data. One way to do this is to fit a polynomial model and check whether it fits the data better than a linear model. There are other reasons, such as theoretical, that indicate to fit a quadratic or higher order model because it is believed that the variables relationship is inherently polynomial in nature.

Let's fit a quadratic model for the `mtcars` dataset. For a linear model see Linear regression on the mtcars dataset.

First we make a scatter plot of the variables `mpg` (Miles/gallon), `disp` (Displacement (cu.in.)), and `wt` (Weight (1000 lbs)). The relationship among `mpg` and `disp` appears non-linear.

```
plot(mtcars[,c("mpg","disp","wt")])
```


www.dbooks.org



A linear fit will show that `disp` is not significant.

```
fit0 = lm(mpg ~ wt+disp, mtcars)
summary(fit0)

# Coefficients:
#             Estimate Std. Error t value Pr(>|t|)
#(Intercept) 34.96055    2.16454  16.151 4.91e-16 ***
#wt          -3.35082    1.16413  -2.878  0.00743 **
#disp        -0.01773    0.00919  -1.929  0.06362 .
#---
#Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
#Residual standard error: 2.917 on 29 degrees of freedom
#Multiple R-squared:  0.7809,    Adjusted R-squared:  0.7658
```

Then, to get the result of a quadratic model, we added `I(disp^2)`. The new model appears better when looking at `R^2` and all variables are significant.

```
fit1 = lm(mpg ~ wt+disp+I(disp^2), mtcars)
summary(fit1)

# Coefficients:
#               Estimate Std. Error t value Pr(>|t|)
#(Intercept) 41.4019837  2.4266906  17.061  2.5e-16 ***
#wt          -3.4179165  0.9545642  -3.581 0.001278 **
#disp        -0.0823950  0.0182460  -4.516 0.000104 ***
#I(disp^2)    0.0001277  0.0000328   3.892 0.000561 ***
#---
```

```
#Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
#Residual standard error: 2.391 on 28 degrees of freedom
#Multiple R-squared:  0.8578,    Adjusted R-squared:  0.8426
```

As we have three variables, the fitted model is a surface represented by:

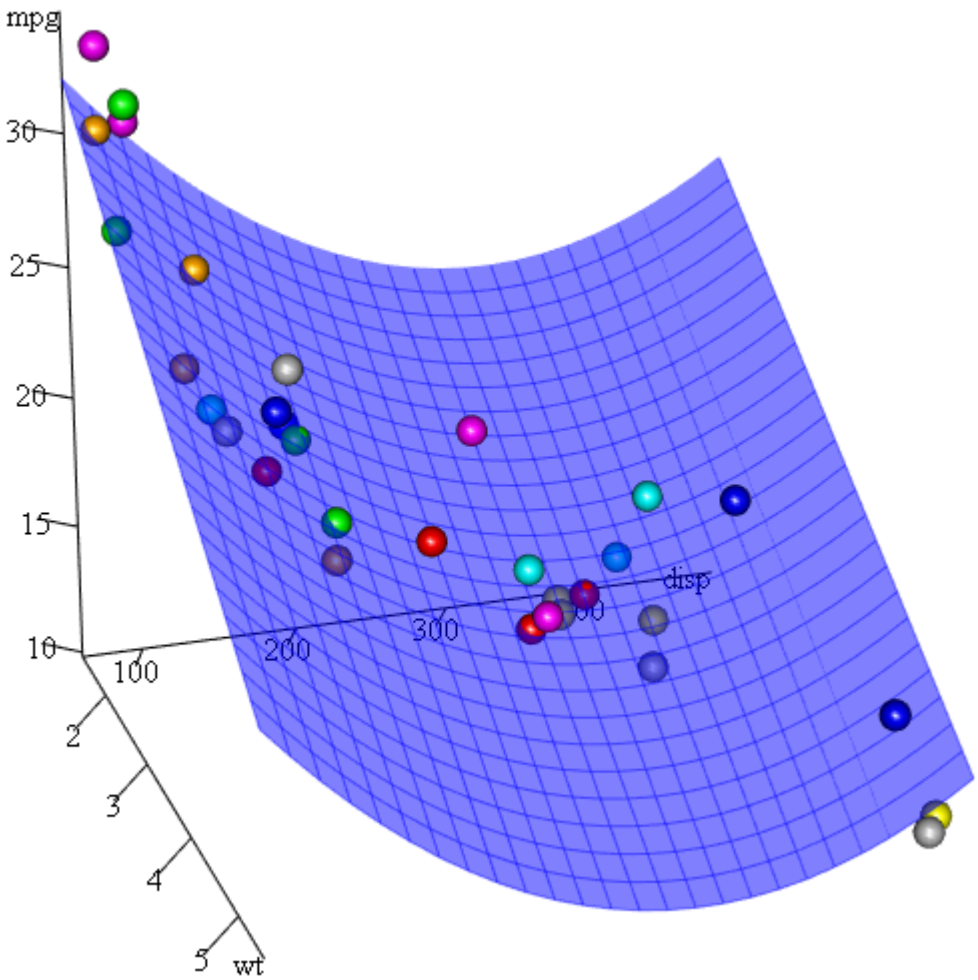
mpg = 41.4020−3.4179*wt−0.0824*disp+0.0001277*disp^2

Another way to specify polynomial regression is using `poly` with parameter `raw=TRUE`, otherwise *orthogonal polynomials* will be considered (see the `help(ploy)` for more information). We get the same result using:

```
summary(lm(mpg ~ wt+poly(disp, 2, raw=TRUE),mtcars))
```

Finally, what if we need to show a plot of the estimated surface? Well there are many options to make `3D` plots in `R`. Here we use `Fit3d` from `p3d`package.

```
library(p3d)
Init3d(family="serif", cex = 1)
Plot3d(mpg ~ disp+wt, mtcars)
Axes3d()
Fit3d(fit1)
```

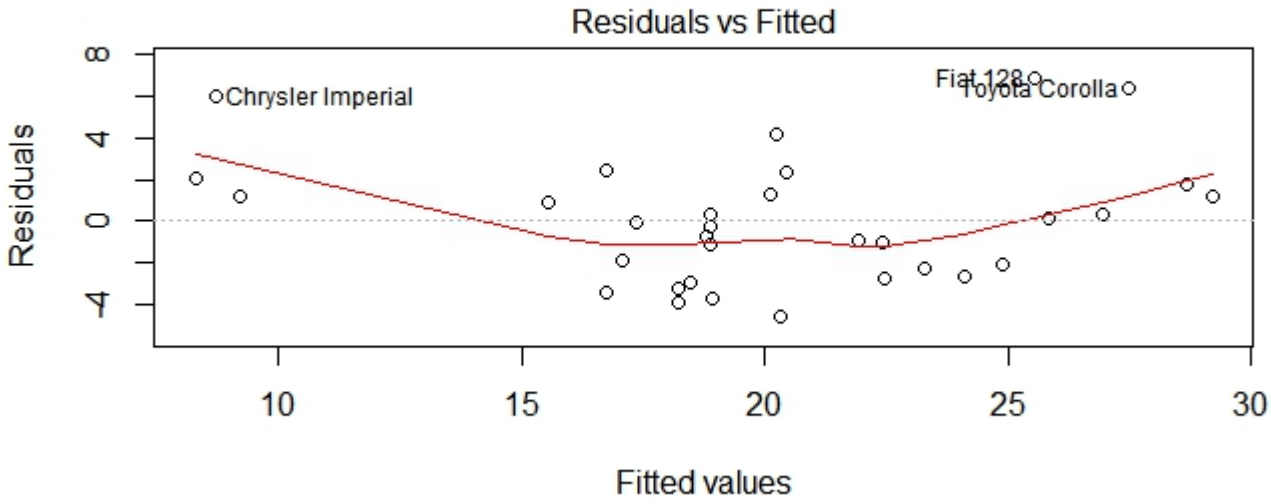


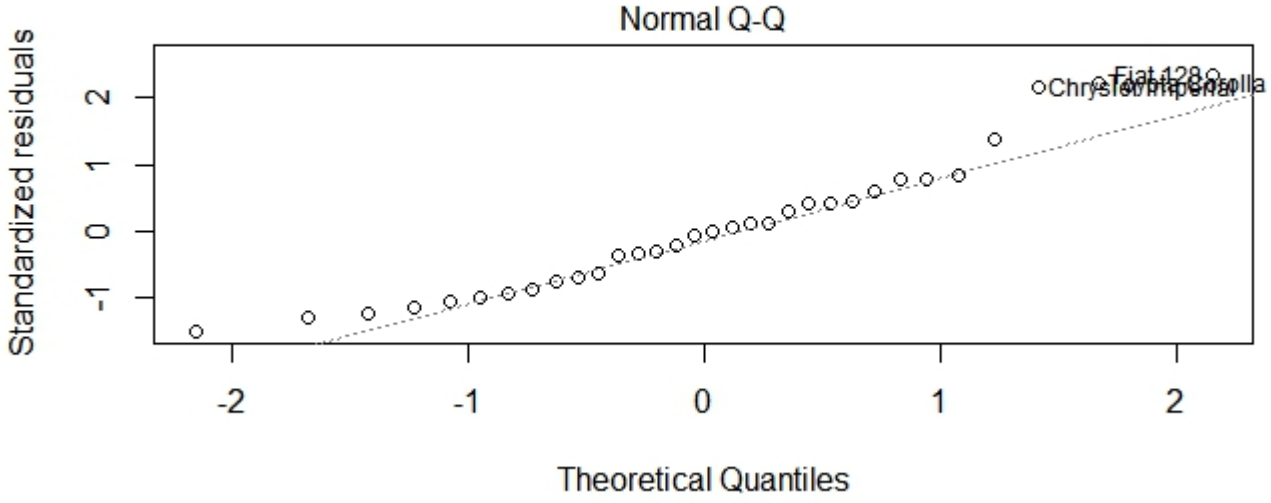
www.dbooks.org

Quality assessment

After building a regression model it is important to check the result and decide if the model is appropriate and works well with the data at hand. This can be done by examining the residuals plot as well as other diagnostic plots.

```
# fit the model
fit <- lm(mpg ~ wt, data = mtcars)
#
par(mfrow=c(2,1))
# plot model object
plot(fit, which =1:2)
```





These plots check for two assumptions that were made while building the model:

1. That the expected value of the predicted variable (in this case `mpg`) is given by a linear

combination of the predictors (in this case `wt`). We expect this estimate to be unbiased. So the residuals should be centered around the mean for all values of the predictors. In this case we see that the residuals tend to be positive at the ends and negative in the middle, suggesting a non-linear relationship between the variables.
2. That the actual predicted variable is normally distributed around its estimate. Thus, the residuals should be normally distributed. For normally distributed data, the points in a normal Q-Q plot should lie on or close to the diagonal. There is some amount of skew at the ends here.

Using the 'predict' function

Once a model is built `predict` is the main function to test with new data. Our example will use the `mtcars` built-in dataset to regress miles per gallon against displacement:

```
my_mdl <- lm(mpg ~ disp, data=mtcars)
my_mdl

Call:
lm(formula = mpg ~ disp, data = mtcars)

Coefficients:
(Intercept)         disp
   29.59985     -0.04122
```

If I had a new data source with displacement I could see the estimated miles per gallon.

```
set.seed(1234)
newdata <- sample(mtcars$disp, 5)
newdata
[1] 258.0  71.1  75.7 145.0 400.0

newdf <- data.frame(disp=newdata)
predict(my_mdl, newdf)
       1        2        3        4        5
18.96635 26.66946 26.47987 23.62366 13.11381
```

The most important part of the process is to create a new data frame with the same column names as the original data. In this case, the original data had a column labeled `disp`, I was sure to call the new data that same name.

Caution

Let's look at a few common pitfalls:

1. not using a data.frame in the new object:

```
predict(my_mdl, newdata)
Error in eval(predvars, data, env) :
   numeric 'envir' arg not of length one
```

2. not using same names in new data frame:

www.dbooks.org

```
newdf2 <- data.frame(newdata)
predict(my_mdl, newdf2)
Error in eval(expr, envir, enclos) : object 'disp' not found
```

Accuracy

To check the accuracy of the prediction you will need the actual y values of the new data. In this example, `newdf` will need a column for 'mpg' and 'disp'.

```
newdf <- data.frame(mpg=mtcars$mpg[1:10], disp=mtcars$disp[1:10])
#     mpg  disp
# 1  21.0 160.0
# 2  21.0 160.0
# 3  22.8 108.0
# 4  21.4 258.0
# 5  18.7 360.0
# 6  18.1 225.0
# 7  14.3 360.0
# 8  24.4 146.7
# 9  22.8 140.8
# 10 19.2 167.6

p <- predict(my_mdl, newdf)

#root mean square error
sqrt(mean((p - newdf$mpg)^2, na.rm=TRUE))
[1] 2.325148
```

Read Linear Models (Regression) online: https://riptutorial.com/r/topic/801/linear-models--regression-

Chapter 62: Lists

Examples

Quick Introduction to Lists

In general, most of the objects you would interact with as a user would tend to be a vector; e.g numeric vector, logical vector. These objects can only take in a single type of variable (a numeric vector can only have numbers inside it).

A list would be able to store any type variable in it, making it to the generic object that can store any type of variables we would need.

Example of initializing a list

```
exampleList1 <- list('a', 'b')
exampleList2 <- list(1, 2)
exampleList3 <- list('a', 1, 2)
```

In order to understand the data that was defined in the list, we can use the str function.

```
str(exampleList1)
str(exampleList2)
str(exampleList3)
```

Subsetting of lists distinguishes between extracting a slice of the list, i.e. obtaining a list containing a subset of the elements in the original list, and extracting a single element. Using the `[` operator commonly used for vectors produces a new list.

```
# Returns List
exampleList3[1]
exampleList3[1:2]
```

To obtain a single element use `[[` instead.

```
# Returns Character
exampleList3[[1]]
```

List entries may be named:

```
exampleList4 <- list(
    num = 1:3,
    numeric = 0.5,
    char = c('a', 'b')
)
```

The entries in named lists can be accessed by their name instead of their index.

www.dbooks.org

```
exampleList4[['char']]
```

Alternatively the `$` operator can be used to access named elements.

```
exampleList4$num
```

This has the advantage that it is faster to type and may be easier to read but it is important to be aware of a potential pitfall. The `$` operator uses partial matching to identify matching list elements and may produce unexpected results.

```
exampleList5 <- exampleList4[2:3]

exampleList4$num
# c(1, 2, 3)

exampleList5$num
# 0.5

exampleList5[['num']]
# NULL
```

Lists can be particularly useful because they can store objects of different lengths and of various classes.

```
## Numeric vector
exampleVector1 <- c(12, 13, 14)
## Character vector
exampleVector2 <- c("a", "b", "c", "d", "e", "f")
## Matrix
exampleMatrix1 <- matrix(rnorm(4), ncol = 2, nrow = 2)
## List
exampleList3 <- list('a', 1, 2)

exampleList6 <- list(
    num = exampleVector1,
    char = exampleVector2,
    mat = exampleMatrix1,
    list = exampleList3
)
exampleList6
#$num
#[1] 12 13 14
#
#$char
#[1] "a" "b" "c" "d" "e" "f"
#
#$mat
#           [,1]        [,2]
#[1,] 0.5013050 -1.88801542
#[2,] 0.4295266  0.09751379
#
#$list
#$list[[1]]
#[1] "a"
#
#$list[[2]]
```

```
#[1] 1
#
#$list[[3]]
#[1] 2
```

Introduction to lists

Lists allow users to store multiple elements (like vectors and matrices) under a single object. You can use the `list` function to create a list:

```
l1 <- list(c(1, 2, 3), c("a", "b", "c"))
l1
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] "a" "b" "c"
```

Notice the vectors that make up the above list are different classes. Lists allow users to group elements of different classes. Each element in a list can also have a name. List names are accessed by the `names` function, and are assigned in the same manner row and column names are assigned in a matrix.

```
names(l1)
## NULL
names(l1) <- c("vector1", "vector2")
l1
## $vector1
## [1] 1 2 3
##
## $vector2
## [1] "a" "b" "c"
```

It is often easier and safer to declare the list names when creating the list object.

```
l2 <- list(vec = c(1, 3, 5, 7, 9),
       mat = matrix(data = c(1, 2, 3), nrow = 3))
l2
## $vec
## [1] 1 3 5 7 9
##
## $mat
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
names(l2)
## [1] "vec" "mat"
```

Above the list has two elements, named "vec" and "mat," a vector and matrix, resepcively.

Reasons for using lists

www.dbooks.org

To the average R user, the list structure may appear to be the one of the more complicated data structures to manipulate. There are no guarantees that all the elements within it are of the same type; There is no guaranteed structure of how complicated/non-complicated that the list would be (An element in a list could be a list)

However, one of the main reasons when to use lists to use it to pass parameters between functions.

```
# Function example which returns a single element numeric vector
exampleFunction1 <- function(num1, num2){
    result <- num1 + num2
    return(result)
}

# Using example function 1
exampleFunction1(1, 2)

# Function example which returns a simple numeric vector
exampleFunction2 <- function(num1, num2, multiplier){
    tempResult1 <- num1 + num2
    tempResult2 <- tempResult1 * multiplier
    result <- c(tempResult1, tempResult2)
    return(result)
}

# Using example function 2
exampleFunction2(1, 2, 4)
```

In the above example, the returned results are just simple numeric vectors. There is no issues to pass over such simple vectors.

It is important to note at this point that generally, R functions only return 1 result at a time (You can use if conditions to return different results). However, if you intend to create a function which takes a set of parameters and returns several type of results such a numeric vector(settings value) and a data frame (from the calculation), you would need to dump all these results in a list before returning it.

```
# We will be using mtcars dataset here
# Function which returns a result that is supposed to contain multiple type of results
# This can be solved by putting the results into a list
exampleFunction3 <- function(dataframe, removeColumn, sumColumn){
    resultDataFrame <- dataframe[, -removeColumn]
    resultSum <- sum(dataframe[, sumColumn])
    resultList <- list(resultDataFrame, resultSum)
    return(resultList)
}

# Using example function 3
exampleResult <- exampleFunction3(mtcars, 2, 4)
exampleResult[[1]]
exampleResult[[2]]
```

Convert a list to a vector while keeping empty list elements

When one wishes to convert a list to a vector or data.frame object empty elements are typically dropped.

This can be problematic which a list is created of a desired length are created with some empty values (e.g. a list with n elements is created to be added to an m x n matrix, data.frame, or data.table). It is possible to losslessly convert a list to a vector however, retaining empty elements:

```
res <- list(character(0), c("Luzhuang", "Laisu", "Peihui"), character(0),
    c("Anjiangping", "Xinzhai", "Yongfeng"), character(0), character(0),
    c("Puji", "Gaotun", "Banjingcun"), character(0), character(0),
    character(0))
res
```

```
[[1]]
character(0)

[[2]]
[1] "Luzhuang" "Laisu"    "Peihui"

[[3]]
character(0)

[[4]]
[1] "Anjiangping" "Xinzhai"     "Yongfeng"

[[5]]
character(0)

[[6]]
character(0)

[[7]]
[1] "Puji"       "Gaotun"     "Banjingcun"

[[8]]
character(0)

[[9]]
character(0)

[[10]]
character(0)
```

```
res <- sapply(res, function(s) if (length(s) == 0) NA_character_ else paste(s, collapse = "
"))
res
```

```
 [1] NA                             "Luzhuang Laisu Peihui"        NA
"Anjiangping Xinzhai Yongfeng" NA

 [6] NA                             "Puji Gaotun Banjingcun"       NA
NA                             NA
```

Serialization: using lists to pass informations

www.dbooks.org

There exist cases in which it is necessary to put data of different types together. In Azure ML for example, it is necessary to pass informations from a R script module to another one exclusively throught dataframes. Suppose we have a dataframe and a number:

```
> df
       name height        team fun_index title age         desc Y
1    Andrea    195       Lazio        97     6  33   eccellente 1
2      Paja    165  Fiorentina        87     6  31       deciso 1
3      Roro    190       Lazio        65     6  28       strano 0
4    Gioele     70       Lazio       100     0   2    simpatico 1
5     Cacio    170    Juventus        81     3  33         duro 0
6     Edola    171       Lazio        72     5  32     svampito 1
7    Salami    175       Inter        75     3  30  doppiopasso 1
8    Braugo    180       Inter        79     5  32          gjn 0
9     Benna    158    Juventus        80     6  28     esaurito 0
10   Riggio    182       Lazio        92     5  31     certezza 1
11 Giordano    185        Roma        79     5  29        buono 1

> number <- "42"
```

We can access to this information:

```
> paste(df$name[4],"is a",df3$team[4], "supporter." )
[1] "Gioele is a  Lazio supporter."
> paste("The answer to THE question is", number )
[1] "The answer to THE question is 42"
```

In order to put different types of data in a dataframe we have to use the list object and the serialization. In particular we have to put the data in a generic list and then put the list in a particular dataframe:

```
l <- list(df,number)
dataframe_container <- data.frame(out2 = as.integer(serialize(l, connection=NULL)))
```

Once we have stored the information in the dataframe, we need to deserialize it in order to use it:

```
#----- unserialize ----------------------------------------+
unser_obj <- unserialize(as.raw(dataframe_container$out2))
#----- taking back the elements---------------------------+
df_mod        <- unser_obj[1][[1]]
number_mod    <- unser_obj[2][[1]]
```

Then, we can verify that the data are transfered correctly:

```
> paste(df_mod$name[4],"is a",df_mod$team[4], "supporter." )
[1] "Gioele is a  Lazio supporter."
> paste("The answer to THE question is", number_mod )
[1] "The answer to THE question is 42"
```

Read Lists online: https://riptutorial.com/r/topic/1365/lists

Chapter 63: lubridate

Syntax

- ymd_hms(..., quiet = FALSE, tz = "UTC", locale = Sys.getlocale("LC_TIME"))
- now(tzone = "")
- interval(start, end, tzone = attr(start, "tzone"))
- duration(num = NULL, units = "seconds", ...)
- period(num = NULL, units = "second", ...)

Remarks

To install package from CRAN:

```
install.packages("lubridate")
```

To install development version from Github:

```
library(devtools)
# dev mode allows testing of development packages in a sandbox, without interfering
# with the other packages you have installed.
dev_mode(on=T)
install_github("hadley/lubridate")
dev_mode(on=F)
```

To get vignettes on lubridate package:

```
vignette("lubridate")
```

To get help about some function `foo`:

```
help(foo)     # help about function foo
?foo          # same thing

# Example
# help("is.period")
# ?is.period
```

To get examples for a function `foo`:

```
example("foo")

# Example
# example("interval")
```

Examples

www.dbooks.org

Parsing dates and datetimes from strings with lubridate

The `lubridate` package provides convenient functions to format date and datetime objects from character strings. The functions are permutations of

Letter	Element to parse	Base R equivalent
y	year	`%y`, `%Y`
m (with y and d)	month	`%m`, `%b`, `%h`, `%B`
d	day	`%d`, `%e`
h	hour	`%H`, `%I%p`
m (with h and s)	minute	`%M`
s	seconds	`%S`

e.g. `ymd()` for parsing a date with the year followed by the month followed by the day, e.g. `"2016-07-22"`, or `ymd_hms()` for parsing a datetime in the order year, month, day, hours, minutes, seconds, e.g. `"2016-07-22 13:04:47"`.

The functions are able to recognize most separators (such as `/`, `-`, and whitespace) without additional arguments. They also work with inconsistent separators.

Dates

The date functions return an object of class `Date`.

```
library(lubridate)

mdy(c(' 07/02/2016 ', '7 / 03 / 2016', ' 7 / 4 / 16 '))
## [1] "2016-07-02" "2016-07-03" "2016-07-04"

ymd(c("20160724","2016/07/23","2016-07-25"))    # inconsistent separators
## [1] "2016-07-24" "2016-07-23" "2016-07-25"
```

Datetimes

Utility functions

Datetimes can be parsed using `ymd_hms` variants including `ymd_hm` and `ymd_h`. All datetime functions can accept a `tz` timezone argument akin to that of `as.POSIXct` or `strptime`, but which defaults to `"UTC"`

instead of the local timezone.

The datetime functions return an object of class `POSIXct`.

```
x <- c("20160724 130102","2016/07/23 14:02:01","2016-07-25 15:03:00")
ymd_hms(x, tz="EST")
## [1] "2016-07-24 13:01:02 EST" "2016-07-23 14:02:01 EST"
## [3] "2016-07-25 15:03:00 EST"

ymd_hms(x)
## [1] "2016-07-24 13:01:02 UTC" "2016-07-23 14:02:01 UTC"
## [3] "2016-07-25 15:03:00 UTC"
```

Parser functions

`lubridate` also includes three functions for parsing datetimes with a formatting string like `as.POSIXct` or `strptime`:

Function	Output Class	Formatting strings accepted
`parse_date_time`	POSIXct	Flexible. Will accept `strptime`-style with `%` or `lubridate` datetime function name style, e.g `"ymd hms"`. Will accept a vector of orders for heterogeneous data and guess which is appropriate.
`parse_date_time2`	Default POSIXct; if `lt = TRUE`, POSIXlt	Strict. Accepts only `strptime` tokens (with or without `%`) from a limited set.
`fast_strptime`	Default POSIXlt; if `lt = FALSE`, POSIXct	Strict. Accepts only `%`-delimited `strptime` tokens with delimiters (`-`, `/`, `:`, etc.) from a limited set.

```
x <- c('2016-07-22 13:04:47', '07/22/2016 1:04:47 pm')

parse_date_time(x, orders = c('mdy Imsp', 'ymd hms'))
## [1] "2016-07-22 13:04:47 UTC" "2016-07-22 13:04:47 UTC"

x <- c('2016-07-22 13:04:47', '2016-07-22 14:47:58')

parse_date_time2(x, orders = 'Ymd HMS')
## [1] "2016-07-22 13:04:47 UTC" "2016-07-22 14:47:58 UTC"

fast_strptime(x, format = '%Y-%m-%d %H:%M:%S')
## [1] "2016-07-22 13:04:47 UTC" "2016-07-22 14:47:58 UTC"
```

`parse_date_time2` and `fast_strptime` use a fast C parser for efficiency.

See `?parse_date_time` for formatting tokens.

Parsing date and time in lubridate

www.dbooks.org

Lubridate provides `ymd()` series of functions for parsing character strings into dates. The letters y, m, and d correspond to the year, month, and day elements of a date-time.

```
mdy("07-21-2016")                 # Returns Date

## [1] "2016-07-21"

mdy("07-21-2016", tz = "UTC")     # Returns a vector of class POSIXt

## "2016-07-21 UTC"

dmy("21-07-2016")                 # Returns Date

## [1] "2016-07-21"

dmy(c("21.07.2016", "22.07.2016")) # Returns vector of class Date

## [1] "2016-07-21" "2016-07-22"
```

Manipulating date and time in lubridate

```
date <- now()
date
## "2016-07-22 03:42:35 IST"

year(date)
## 2016

minute(date)
## 42

wday(date, label = T, abbr = T)
# [1] Fri
# Levels: Sun < Mon < Tues < Wed < Thurs < Fri < Sat

day(date) <- 31
## "2016-07-31 03:42:35 IST"

# If an element is set to a larger value than it supports, the difference
#  will roll over into the next higher element
day(date) <- 32
## "2016-08-01 03:42:35 IST"
```

Instants

An instant is a specific moment in time. Any date-time object that refers to a moment of time is recognized as an instant. To test if an object is an instant, use `is.instant`.

```
library(lubridate)

today_start <- dmy_hms("22.07.2016 12:00:00", tz = "IST") # default tz="UTC"
today_start
## [1] "2016-07-22 12:00:00 IST"
is.instant(today_start)
## [1] TRUE
```

```
now_dt <- ymd_hms(now(), tz="IST")
now_dt
## [1] "2016-07-22 13:53:09 IST"
is.instant(now_dt)
## [1] TRUE

is.instant("helloworld")
## [1] FALSE
is.instant(60)
## [1] FALSE
```

Intervals, Durations and Periods

Intervals are simplest way of recording timespans in lubridate. An interval is a span of time that occurs between two specific **instants**.

```
# create interval by substracting two instants
today_start <- ymd_hms("2016-07-22 12-00-00", tz="IST")
today_start
## [1] "2016-07-22 12:00:00 IST"
today_end <- ymd_hms("2016-07-22 23-59-59", tz="IST")
today_end
## [1] "2016-07-22 23:59:59 IST"
span <- today_end - today_start
span
## Time difference of 11.99972 hours
as.interval(span, today_start)
## [1] 2016-07-22 12:00:00 IST--2016-07-22 23:59:59 IST

# create interval using interval() function
span <- interval(today_start, today_end)
[1] 2016-07-22 12:00:00 IST--2016-07-22 23:59:59 IST
```

Durations measure the exact amount of time that occurs between two instants.

```
duration(60, "seconds")
## [1] "60s"

duration(2, "minutes")
## [1] "120s (~2 minutes)"
```

Note: Units larger than weeks are not used due to their variability.

Durations can be created using `dseconds`, `dminutes` and other duration helper functions.
Run `?quick_durations` for complete list.

```
dseconds(60)
## [1] "60s"

dhours(2)
## [1] "7200s (~2 hours)"

dyears(1)
## [1] "31536000s (~365 days)"
```


www.dbooks.org

Durations can be subtracted and added to instants to get new instants.

```
today_start + dhours(5)
## [1] "2016-07-22 17:00:00 IST"

today_start + dhours(5) + dminutes(30) + dseconds(15)
## [1] "2016-07-22 17:30:15 IST"
```

Durations can be created from intervals.

```
as.duration(span)
[1] "43199s (~12 hours)"
```

Periods measure the change in clock time that occurs between two instants.

Periods can be created using `period` function as well other helper functions like `seconds`, `hours`, etc. To get a complete list of period helper functions, Run `?quick_periods`.

```
period(1, "hour")
## [1] "1H 0M 0S"

hours(1)
## [1] "1H 0M 0S"

period(6, "months")
## [1] "6m 0d 0H 0M 0S"

months(6)
## [1] "6m 0d 0H 0M 0S"

years(1)
## [1] "1y 0m 0d 0H 0M 0S"
```

`is.period` function can be used to check if an object is a period.

```
is.period(years(1))
## [1] TRUE

is.period(dyears(1))
## [1] FALSE
```

Rounding dates

```
now_dt <- ymd_hms(now(), tz="IST")
now_dt
## [1] "2016-07-22 13:53:09 IST"
```

`round_date()` takes a date-time object and rounds it to the nearest integer value of the specified time unit.

```
round_date(now_dt, "minute")
## [1] "2016-07-22 13:53:00 IST"
```

```
round_date(now_dt, "hour")
## [1] "2016-07-22 14:00:00 IST"

round_date(now_dt, "year")
## [1] "2017-01-01 IST"
```

`floor_date()` takes a date-time object and rounds it down to the nearest integer value of the specified time unit.

```
floor_date(now_dt, "minute")
## [1] "2016-07-22 13:53:00 IST"

floor_date(now_dt, "hour")
## [1] "2016-07-22 13:00:00 IST"

floor_date(now_dt, "year")
## [1] "2016-01-01 IST"
```

`ceiling_date()` takes a date-time object and rounds it up to the nearest integer value of the specified time unit.

```
ceiling_date(now_dt, "minute")
## [1] "2016-07-22 13:54:00 IST"

ceiling_date(now_dt, "hour")
## [1] "2016-07-22 14:00:00 IST"

ceiling_date(now_dt, "year")
## [1] "2017-01-01 IST"
```

Difference between period and duration

Unlike durations, periods can be used to accurately model clock times without knowing when events such as leap seconds, leap days, and DST changes occur.

```
start_2012 <- ymd_hms("2012-01-01 12:00:00")
## [1] "2012-01-01 12:00:00 UTC"

# period() considers leap year calculations.
start_2012 + period(1, "years")
## [1] "2013-01-01 12:00:00 UTC"

# Here duration() doesn't consider leap year calculations.
start_2012 + duration(1)
## [1] "2012-12-31 12:00:00 UTC"
```

Time Zones

`with_tz` returns a date-time as it would appear in a different time zone.

```
nyc_time <- now("America/New_York")
nyc_time
```


www.dbooks.org

```
## [1] "2016-07-22 05:49:08 EDT"

# corresponding Europe/Moscow time
with_tz(nyc_time, tzone = "Europe/Moscow")
## [1] "2016-07-22 12:49:08 MSK"
```

`force_tz` returns a the date-time that has the same clock time as x in the new time zone.

```
nyc_time <- now("America/New_York")
nyc_time
## [1] "2016-07-22 05:49:08 EDT"

force_tz(nyc_time, tzone = "Europe/Moscow") # only timezone changes
## [1] "2016-07-22 05:49:08 MSK"
```

Read lubridate online: https://riptutorial.com/r/topic/2496/lubridate

Chapter 64: Machine learning

Examples

Creating a Random Forest model

One example of machine learning algorithms is the Random Forest alogrithm (Breiman, L. (2001). Random Forests. *Machine Learning 45(5)*, p. 5-32). This algorithm is implemented in R according to Breiman's original Fortran implementation in the `randomForest` package.

Random Forest classifier objects can be created in R by preparing the class variable as `factor`, which is already apparent in the `iris` data set. Therefore we can easily create a Random Forest by:

```
library(randomForest)

rf <- randomForest(x = iris[, 1:4],
                   y = iris$Species,
                   ntree = 500,
                   do.trace = 100)

rf

# Call:
#   randomForest(x = iris[, 1:4], y = iris$Species, ntree = 500,      do.trace = 100)
# Type of random forest: classification
# Number of trees: 500
# No. of variables tried at each split: 2
#
# OOB estimate of  error rate: 4%
# Confusion matrix:
#   setosa versicolor virginica class.error
# setosa         50          0         0        0.00
# versicolor      0         47         3        0.06
# virginica       0          3        47        0.06
```

parameters	Description
x	a data frame holding the describing variables of the classes
y	the classes of the individual obserbations. If this vector is `factor`, a classification model is created, if not a regression model is created.
ntree	The number of individual CART trees built
do.trace	every *ith* step, the out-of-the-box errors overall and for each class are returned

Read Machine learning online: https://riptutorial.com/r/topic/8326/machine-learning

www.dbooks.org

Chapter 65: Matrices

Introduction

Matrices store data

Examples

Creating matrices

Under the hood, a matrix is a special kind of vector with two dimensions. Like a vector, a matrix can only have one data class. You can create matrices using the `matrix` function as shown below.

```
matrix(data = 1:6, nrow = 2, ncol = 3)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

As you can see this gives us a matrix of all numbers from 1 to 6 with two rows and three columns. The `data` parameter takes a vector of values, `nrow` specifies the number of rows in the matrix, and `ncol` specifies the number of columns. By convention the matrix is filled by column. The default behavior can be changed with the `byrow` parameter as shown below:

```
matrix(data = 1:6, nrow = 2, ncol = 3, byrow = TRUE)
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Matrices do not have to be numeric – any vector can be transformed into a matrix. For example:

```
matrix(data = c(TRUE, TRUE, TRUE, FALSE, FALSE, FALSE), nrow = 3, ncol = 2)
##      [,1]  [,2]
## [1,] TRUE FALSE
## [2,] TRUE FALSE
## [3,] TRUE FALSE
matrix(data = c("a", "b", "c", "d", "e", "f"), nrow = 3, ncol = 2)
##      [,1] [,2]
## [1,] "a"  "d"
## [2,] "b"  "e"
## [3,] "c"  "f"
```

Like vectors matrices can be stored as variables and then called later. The rows and columns of a matrix can have names. You can look at these using the functions `rownames` and `colnames`. As shown below, the rows and columns don't initially have names, which is denoted by `NULL`. However, you can assign values to them.

```
mat1 <- matrix(data = 1:6, nrow = 2, ncol = 3, byrow = TRUE)
rownames(mat1)
```

```
## NULL
colnames(mat1)
## NULL
rownames(mat1) <- c("Row 1", "Row 2")
colnames(mat1) <- c("Col 1", "Col 2", "Col 3")
mat1
##       Col 1 Col 2 Col 3
## Row 1     1     2     3
## Row 2     4     5     6
```

It is important to note that similarly to vectors, matrices can only have one data type. If you try to specify a matrix with multiple data types the data will be coerced to the higher order data class.

The `class`, `is`, and `as` functions can be used to check and coerce data structures in the same way they were used on the vectors in class 1.

```
class(mat1)
## [1] "matrix"
is.matrix(mat1)
## [1] TRUE
as.vector(mat1)
## [1] 1 4 2 5 3 6
```

Read Matrices online: https://riptutorial.com/r/topic/9019/matrices

www.dbooks.org

Chapter 66: Meta: Documentation Guidelines

Remarks

To discuss editing the R tag Docs, visit the R chat.

Examples

Making good examples

Most of the guidance for creating good examples for Q&A carries over into the documentation.

- Make it minimal and get to the point. Complications and digressions are counterproductive.
- Include both working code and prose explaining it. Neither one is sufficient on its own.
- Don't rely on external sources for data. Generate data or use the datasets library if possible:

```
library(help = "datasets")
```

There are some additional considerations in the context of Docs:

- Refer to built-in docs like `?data.frame` whenever relevant. The SO Docs are not an attempt to replace the built-in docs. It is important to make sure new R users know that the built-in docs exist as well as how to find them.
- Move content that applies to multiple examples to the Remarks section.

Style

Prompts

If you want your code to be copy-pastable, remove prompts such as `R>`, `>`, or `+` at the beginning of each new line. Some Docs authors prefer to not make copy-pasting easy, and that is okay.

Console output

Console output should be clearly distinguished from code. Common approaches include:

- Include prompts on input (as seen when using the console).
- Comment out all output, with `#` or `##` starting each line.
- Print as-is, trusting the leading `[1]` to make the output stand out from the input.
- Add a blank line between code and console output.

Assignment

`=` and `<-` are fine for assigning R objects. Use white space appropriately to avoid writing code that is difficult to parse, such as `x<-1` (ambiguous between `x <- 1` and `x < -1`)

Code comments

Be sure to explain the purpose and function of the code itself. There isn't any hard-and-fast rule on whether this explanation should be in prose or in code comments. Prose may be more readable and allows for longer explanations, but code comments make for easier copy-pasting. Keep both options in mind.

Sections

Many examples are short enough to not need sections, but if you use them, start with H1.

Read Meta: Documentation Guidelines online: https://riptutorial.com/r/topic/5410/meta--documentation-guidelines

www.dbooks.org

Chapter 67: Missing values

Introduction

When we don't know the value a variable takes, we say its value is missing, indicated by `NA`.

Remarks

Missing values are represented by the symbol `NA` (not available). Impossible values (e.g., as a result of `sqrt(-1)`) are represented by the symbol `NaN` (not a number).

Examples

Examining missing data

`anyNA` reports whether any missing values are present; while `is.na` reports missing values elementwise:

```
vec <- c(1, 2, 3, NA, 5)

anyNA(vec)
# [1] TRUE
is.na(vec)
# [1] FALSE FALSE FALSE  TRUE FALSE
```

`is.na` returns a logical vector that is coerced to integer values under arithmetic operations (with FALSE=0, TRUE=1). We can use this to find out how many missing values there are:

```
sum(is.na(vec))
# [1] 1
```

Extending this approach, we can use `colSums` and `is.na` on a data frame to count NAs per column:

```
colSums(is.na(airquality))
#   Ozone Solar.R    Wind    Temp   Month     Day
#      37       7       0       0       0       0
```

The naniar package (currently on github but not CRAN) offers further tools for exploring missing values.

Reading and writing data with NA values

When reading tabular datasets with the `read.*` functions, R automatically looks for missing values that look like `"NA"`. However, missing values are not always represented by `NA`. Sometimes a dot (`.`), a hyphen(`-`) or a character-value (e.g.: `empty`) indicates that a value is `NA`. The `na.strings` parameter of the `read.*` function can be used to tell R which symbols/characters need to be

treated as `NA` values:

```
read.csv("name_of_csv_file.csv", na.strings = "-")
```

It is also possible to indicate that more than one symbol needs to be read as `NA`:

```
read.csv('missing.csv', na.strings = c('.','-'))
```

Similarly, `NA`s can be written with customized strings using the `na` argument to `write.csv`. Other tools for reading and writing tables have similar options.

Using NAs of different classes

The symbol `NA` is for a `logical` missing value:

```
class(NA)
#[1] "logical"
```

This is convenient, since it can easily be coerced to other atomic vector types, and is therefore usually the only `NA` you will need:

```
x <- c(1, NA, 1)
class(x[2])
#[1] "numeric"
```

If you do need a single `NA` value of another type, use `NA_character_`, `NA_integer_`, `NA_real_` or `NA_complex_`. For missing values of fancy classes, subsetting with `NA_integer_` usually works; for example, to get a missing-value Date:

```
class(Sys.Date()[NA_integer_])
# [1] "Date"
```

TRUE/FALSE and/or NA

`NA` is a logical type and a logical operator with an `NA` will return `NA` if the outcome is ambiguous. Below, `NA OR TRUE` evaluates to `TRUE` because at least one side evaluates to `TRUE`, however `NA OR FALSE` returns `NA` because we do not know whether `NA` would have been `TRUE` or `FALSE`

```
NA | TRUE
# [1] TRUE
# TRUE | TRUE is TRUE and FALSE | TRUE is also TRUE.

NA | FALSE
# [1] NA
# TRUE | FALSE is TRUE but FALSE | FALSE is FALSE.

NA & TRUE
# [1] NA
# TRUE & TRUE is TRUE but FALSE & TRUE is FALSE.
```


www.dbooks.org

```
NA & FALSE
# [1] FALSE
# TRUE & FALSE is FALSE and FALSE & FALSE is also FALSE.
```

These properties are helpful if you want to subset a data set based on some columns that contain `NA`.

```
df <- data.frame(v1=0:9,
                 v2=c(rep(1:2, each=4), NA, NA),
                 v3=c(NA, letters[2:10]))

df[df$v2 == 1 & !is.na(df$v2), ]
#  v1 v2   v3
#1  0  1 <NA>
#2  1  1    b
#3  2  1    c
#4  3  1    d

df[df$v2 == 1, ]
     v1 v2   v3
#1     0  1 <NA>
#2     1  1    b
#3     2  1    c
#4     3  1    d
#NA   NA NA <NA>
#NA.1 NA NA <NA>
```

Omitting or replacing missing values

Recoding missing values

Regularly, missing data isn't coded as `NA` in datasets. In SPSS for example, missing values are often represented by the value `99`.

```
num.vec <- c(1, 2, 3, 99, 5)
num.vec
## [1]  1  2  3 99  5
```

It is possible to directly assign NA using subsetting

```
num.vec[num.vec == 99] <- NA
```

However, the preferred method is to use `is.na<-` as below. The help file (`?is.na`) states:

> `is.na<-` may provide a safer way to set missingness. It behaves differently for factors, for example.

```
is.na(num.vec) <- num.vec == 99
```

Both methods return

```
num.vec
## [1]  1  2  3 NA  5
```

Removing missing values

Missing values can be removed in several ways from a vector:

```
num.vec[!is.na(num.vec)]
num.vec[complete.cases(num.vec)]
na.omit(num.vec)
## [1] 1 2 3 5
```

Excluding missing values from calculations

When using arithmetic functions on vectors with missing values, a missing value will be returned:

```
mean(num.vec) # returns: [1] NA
```

The `na.rm` parameter tells the function to exclude the `NA` values from the calculation:

```
mean(num.vec, na.rm = TRUE) # returns: [1] 2.75

# an alternative to using 'na.rm = TRUE':
mean(num.vec[!is.na(num.vec)]) # returns: [1] 2.75
```

Some R functions, like `lm`, have a `na.action` parameter. The default-value for this is `na.omit`, but with `options(na.action = 'na.exclude')` the default behavior of R can be changed.

If it is not necessary to change the default behavior, but for a specific situation another `na.action` is needed, the `na.action` parameter needs to be included in the function call, e.g.:

```
lm(y2 ~ y1, data = anscombe, na.action = 'na.exclude')
```

Read Missing values online: https://riptutorial.com/r/topic/3388/missing-values

www.dbooks.org

Chapter 68: Modifying strings by substitution

Introduction

`sub` and `gsub` are used to edit strings using patterns. See Pattern Matching and Replacement for more on related functions and Regular Expressions for how to build a pattern.

Examples

Rearrange character strings using capture groups

If you want to change the order of a character strings you can use parentheses in the `pattern` to group parts of the string together. These groups can in the `replacement` argument be addresed using consecutive numbers.

The following example shows how you can reorder a vector of names of the form "surname, forename" into a vector of the form "forename surname".

```
library(randomNames)
set.seed(1)

strings <- randomNames(5)
strings
# [1] "Sigg, Zachary"        "Holt, Jake"           "Ortega, Sandra"       "De La Torre,
Nichole"
# [5] "Perkins, Donovon"

sub("^(.+),\\s(.+)$", "\\2 \\1", strings)
# [1] "Zachary Sigg"        "Jake Holt"           "Sandra Ortega"       "Nichole De La Torre"
# [5] "Donovon Perkins"
```

If you only need the surname you could just address the first pairs of parentheses.

```
sub("^(.+),\\s(.+)", "\\1", strings)
# [1] "Sigg"        "Holt"        "Ortega"      "De La Torre" "Perkins"
```

Eliminate duplicated consecutive elements

Let's say we want to eliminate duplicated subsequence element from a string (it can be more than one). For example:

```
2,14,14,14,19
```

and convert it into:

```
2,14,19
```

Using `gsub`, we can achieve it:

```
gsub("(\\d+)(,\\1)+","\\1", "2,14,14,14,19")
[1] "2,14,19"
```

It works also for more than one different repetition, for example:

```
 > gsub("(\\d+)(,\\1)+", "\\1", "2,14,14,14,19,19,20,21")
[1] "2,14,19,20,21"
```

Let's explain the regular expression:

1. `(\\d+)`: A group 1 delimited by () and finds any digit (at least one). Remember we need to use the double backslash (`\\`) here because for a character variable a backslash represents special escape character for literal string delimiters (`\"` or `\'`). `\d\` is equivalent to: `[0-9]`.
2. `,`: A punctuation sign: `,` (we can include spaces or any other delimiter)
3. `\\1`: An identical string to the group 1, i.e.: the repeated number. If that doesn't happen, then the pattern doesn't match.

Let's try a similar situation: eliminate consecutive repeated words:

```
one,two,two,three,four,four,five,six
```

Then, just replace `\d` by `\w`, where `\w` matches any word character, including: any letter, digit or underscore. It is equivalent to `[a-zA-Z0-9_]`:

```
> gsub("(\\w+)(,\\1)+", "\\1", "one,two,two,three,four,four,five,six")
[1] "one,two,three,four,five,six"
>
```

Then, the above pattern includes as a particular case duplicated digits case.

Read Modifying strings by substitution online: https://riptutorial.com/r/topic/9219/modifying-strings-by-substitution

www.dbooks.org

Chapter 69: Natural language processing

Introduction

Natural language processing (NLP) is the field of computer sciences focused on retrieving information from textual input generated by human beings.

Examples

Create a term frequency matrix

The simplest approach to the problem (and the most commonly used so far) is to split sentences into *tokens*. Simplifying, *words* have abstract and subjective meanings to the people using and receiving them, *tokens* have an objective interpretation: an ordered sequence of characters (or bytes). Once sentences are split, the order of the token is disregarded. This approach to the problem in known as **bag of words** model.

A **term frequency** is a dictionary, in which to each token is assigned a *weight*. In the first example, we construct a term frequency matrix from a corpus **corpus** (a collection of **documents**) with the R package `tm`.

```
require(tm)
doc1 <- "drugs hospitals doctors"
doc2 <- "smog pollution environment"
doc3 <- "doctors hospitals healthcare"
doc4 <- "pollution environment water"
corpus <- c(doc1, doc2, doc3, doc4)
tm_corpus <- Corpus(VectorSource(corpus))
```

In this example, we created a corpus of class `Corpus` defined by the package `tm` with two functions `Corpus` and `VectorSource`, which returns a `VectorSource` object from a character vector. The object `tm_corpus` is a list our documents with additional (and optional) metadata to describe each document.

```
str(tm_corpus)
List of 4
 $ 1:List of 2
  ..$ content: chr "drugs hospitals doctors"
  ..$ meta   :List of 7
  .. ..$ author       : chr(0)
  .. ..$ datetimestamp: POSIXlt[1:1], format: "2017-06-03 00:31:34"
  .. ..$ description  : chr(0)
  .. ..$ heading      : chr(0)
  .. ..$ id           : chr "1"
  .. ..$ language     : chr "en"
  .. ..$ origin       : chr(0)
  .. ..- attr(*, "class")= chr "TextDocumentMeta"
  ..- attr(*, "class")= chr [1:2] "PlainTextDocument" "TextDocument"
[truncated]
```

Once we have a `Corpus`, we can proceed to preprocess the tokens contained in the `Corpus` to improve the quality of the final output (the term frequency matrix). To do this we use the `tm` function `tm_map`, which similarly to the `apply` family of functions, transform the documents in the corpus by applying a function to each document.

```
tm_corpus <- tm_map(tm_corpus, tolower)
tm_corpus <- tm_map(tm_corpus, removeWords, stopwords("english"))
tm_corpus <- tm_map(tm_corpus, removeNumbers)
tm_corpus <- tm_map(tm_corpus, PlainTextDocument)
tm_corpus <- tm_map(tm_corpus, stemDocument, language="english")
tm_corpus <- tm_map(tm_corpus, stripWhitespace)
tm_corpus <- tm_map(tm_corpus, PlainTextDocument)
```

Following these transformations, we finally create the term frequency matrix with

```
tdm <- TermDocumentMatrix(tm_corpus)
```

which gives a

```
<<TermDocumentMatrix (terms: 8, documents: 4)>>
Non-/sparse entries: 12/20
Sparsity           : 62%
Maximal term length: 9
Weighting          : term frequency (tf)
```

that we can view by transforming it to a matrix

```
as.matrix(tdm)

           Docs
Terms       character(0) character(0) character(0) character(0)
  doctor               1            0            1            0
  drug                 1            0            0            0
  environ              0            1            0            1
  healthcar            0            0            1            0
  hospit               1            0            1            0
  pollut               0            1            0            1
  smog                 0            1            0            0
  water                0            0            0            1
```

Each row represents the frequency of each token - that as you noticed have been stemmed (e.g. `environment` to `environ`) - in each document (4 documents, 4 columns).

In the previous lines, we have weighted each pair token/document with the absolute frequency (i.e. the number of instances of the token that appear in the document).

Read Natural language processing online: https://riptutorial.com/r/topic/10119/natural-language-processing

www.dbooks.org

Chapter 70: Network analysis with the igraph package

Examples

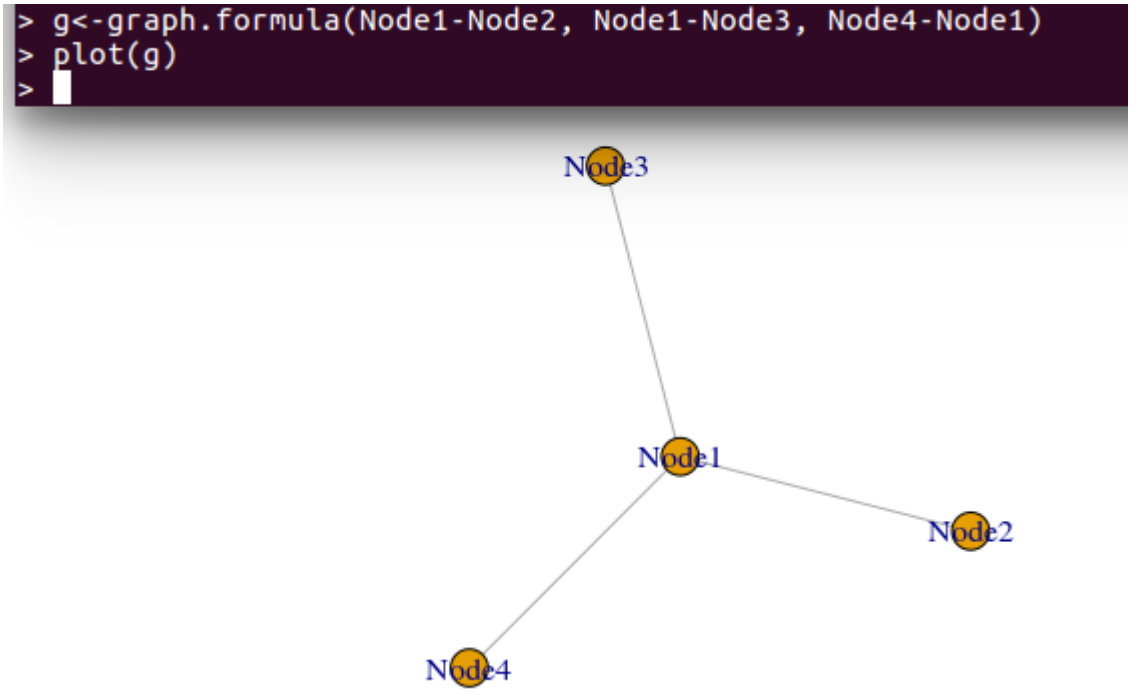
Simple Directed and Non-directed Network Graphing

The igraph package for R is a wonderful tool that can be used to model networks, both real and virtual, with simplicity. This example is meant to demonstrate how to create two simple network graphs using the igraph package within R v.3.2.3.

Non-Directed Network

The network is created with this piece of code:

```
g<-graph.formula(Node1-Node2, Node1-Node3, Node4-Node1)
plot(g)
```

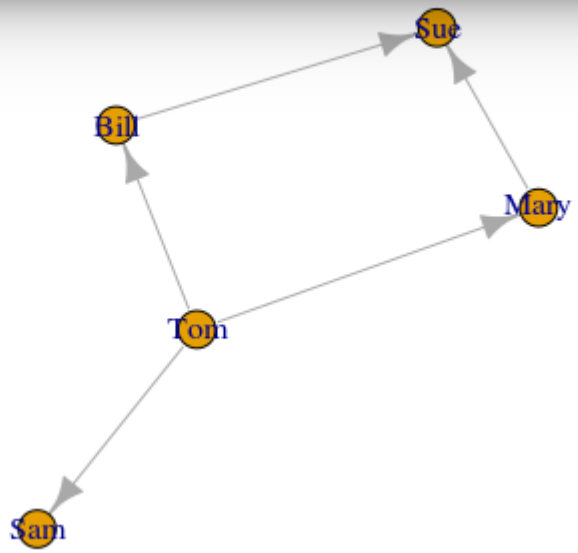


Directed Network

```
dg<-graph.formula(Tom-+Mary, Tom-+Bill, Tom-+Sam, Sue+-Mary, Bill-+Sue)
plot(dg)
```

This code will then generate a network with arrows:

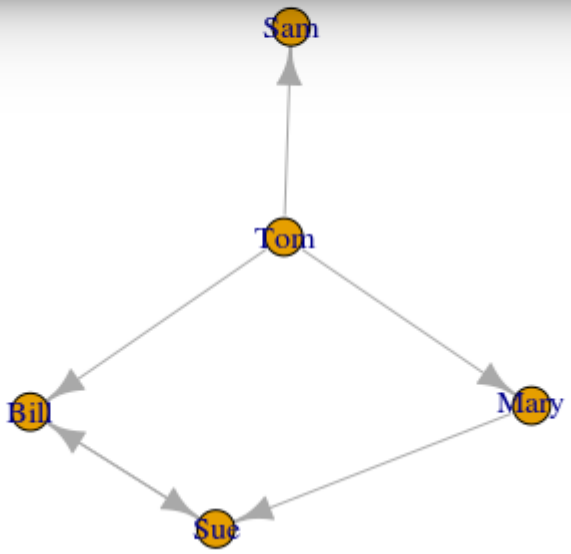
```
> dg<-graph.formula(Tom-+Mary, Tom-+Bill, Tom-+Sam, Sue+-Mary, Bill-+Sue)
> plot(dg)
>
```



Code example of how to make a double sided arrow:

```
dg<-graph.formula(Tom-+Mary, Tom-+Bill, Tom-+Sam, Sue+-Mary, Bill++Sue)
plot(dg)
```

```
> dg<-graph.formula(Tom-+Mary, Tom-+Bill, Tom-+Sam, Sue+-Mary, Bill++Sue)
> plot(dg)
>
```



Read Network analysis with the igraph package online: https://riptutorial.com/r/topic/4851/network-analysis-with-the-igraph-package

www.dbooks.org

Chapter 71: Non-standard evaluation and standard evaluation

Introduction

Dplyr and many modern libraries in R use non-standard evaluation (NSE) for interactive programming and standard evaluation (SE) for programming1.

For instance, the `summarise()` function use non-standard evaluation but relies on the `summarise_()` which uses standard evaluation.

The lazyeval library makes it easy to turn standard evaluation function into NSE functions.

Examples

Examples with standard dplyr verbs

NSE functions should be used in interactive programming. However, when developping new functions in a new package, it's better to use SE version.

Load dplyr and lazyeval :

```
library(dplyr)
library(lazyeval)
```

Filtering

NSE version

```
filter(mtcars, cyl == 8)
filter(mtcars, cyl < 6)
filter(mtcars, cyl < 6 & vs == 1)
```

SE version (to be use when programming functions in a new package)

```
filter_(mtcars, .dots = list(~ cyl == 8))
filter_(mtcars, .dots = list(~ cyl < 6))
filter_(mtcars, .dots = list(~ cyl < 6, ~ vs == 1))
```

Summarise

NSE version

```
summarise(mtcars,  mean(disp))
summarise(mtcars,  mean_disp = mean(disp))
```

SE version

```
summarise_(mtcars, .dots = lazyeval::interp(~ mean(x), x = quote(disp)))
summarise_(mtcars, .dots = setNames(list(lazyeval::interp(~ mean(x), x = quote(disp))),
"mean_disp"))
summarise_(mtcars, .dots = list("mean_disp" = lazyeval::interp(~ mean(x), x = quote(disp))))
```

Mutate

NSE version

```
mutate(mtcars, displ_l = disp / 61.0237)
```

SE version

```
mutate_(
    .data = mtcars,
    .dots = list(
        "displ_l" = lazyeval::interp(
                        ~ x / 61.0237, x = quote(disp)
            )
         )
)
```

Read Non-standard evaluation and standard evaluation online:
https://riptutorial.com/r/topic/9365/non-standard-evaluation-and-standard-evaluation

www.dbooks.org

Chapter 72: Numeric classes and storage modes

Examples

Numeric

Numeric represents integers and doubles and is the default mode assigned to vectors of numbers. The function `is.numeric()` will evaluate whether a vector is numeric. It is important to note that although integers and doubles will pass `is.numeric()`, the function `as.numeric()` will always attempt to convert to type double.

```
x <- 12.3
y <- 12L

#confirm types
typeof(x)
[1] "double"
typeof(y)
[1] "integer"

# confirm both numeric
is.numeric(x)
[1] TRUE
is.numeric(y)
[1] TRUE

# logical to numeric
as.numeric(TRUE)
[1] 1

# While TRUE == 1, it is a double and not an integer
is.integer(as.numeric(TRUE))
[1] FALSE
```

Doubles are R's default numeric value. They are double precision vectors, meaning that they take up 8 bytes of memory for each value in the vector. R has no single precision data type and so all real numbers are stored in the double precision format.

```
is.double(1)
TRUE
is.double(1.0)
TRUE
is.double(1L)
FALSE
```

Integers are whole numbers that can be written without a fractional component. Integers are represented by a number with an L after it. Any number without an L after it will be considered a double.

```
typeof(1)
[1] "double"
class(1)
[1] "numeric"
typeof(1L)
[1] "integer"
class(1L)
[1] "integer"
```

Though in most cases using an integer or double will not matter, sometimes replacing doubles with integers will consume less memory and operational time. A double vector uses 8 bytes per element while an integer vector uses only 4 bytes per element. As the size of vectors increases, using proper types can dramatically speed up processes.

```
#  test speed on lots of arithmetic
microbenchmark(
  for( i in 1:100000){
  2L * i
  10L + i
},

for( i in 1:100000){
  2.0 * i
  10.0 + i
}
)
Unit: milliseconds
                                          expr      min       lq     mean   median       uq
max neval
 for (i in 1:1e+05) {     2L * i     10L + i } 40.74775 42.34747 50.70543 42.99120 65.46864
94.11804   100
   for (i in 1:1e+05) {     2 * i     10 + i } 41.07807 42.38358 53.52588 44.26364 65.84971
83.00456   100
```

Read Numeric classes and storage modes online: https://riptutorial.com/r/topic/9018/numeric-classes-and-storage-modes

www.dbooks.org

Chapter 73: Object-Oriented Programming in R

Introduction

This documentation page describes the four object systems in R and their high-level similarities and differences. Greater detail on each individual system can be found on its own topic page.

The four systems are: S3, S4, Reference Classes, and S6.

Examples

S3

The S3 object system is a very simple OO system in R.

Every object has an S3 class. It can be get (got?) with the function `class`.

```
> class(3)
[1] "numeric"
```

It can also be set with the function `class`:

```
> bicycle <- 2
> class(bicycle) <- 'vehicle'
> class(bicycle)
[1] "vehicle"
```

It can also be set with the function `attr`:

```
> velocipede <- 2
> attr(velocipede, 'class') <- 'vehicle'
> class(velocipede)
[1] "vehicle"
```

An object can have many classes:

```
> class(x = bicycle) <- c('human-powered vehicle', class(x = bicycle))
> class(x = bicycle)
[1] "human-powered vehicle" "vehicle"
```

When using a generic function, R uses the first element of the class that has an available generic.

For example:

```
> summary.vehicle <- function(object, ...) {
```

```
+   message('this is a vehicle')
+ }
> summary(object = my_bike)
this is a vehicle
```

But if we now define a `summary.bicycle`:

```
> summary.bicycle <- function(object, ...) {
+   message('this is a bicycle')
+ }
> summary(object = my_bike)
this is a bicycle
```

Read Object-Oriented Programming in R online: https://riptutorial.com/r/topic/9723/object-oriented-programming-in-r

www.dbooks.org

Chapter 74: Parallel processing

Remarks

Parallelization on remote machines require libraries to be downloaded on each machine. Prefer `package::function()` calls. Several packages have parallelization natively built-in, including `caret`, `pls` and `plyr`.

Microsoft R Open (Revolution R) also uses multi-threaded BLAS/LAPACK libraries which intrinsically parallelizes many common functions.

Examples

Parallel processing with foreach package

The `foreach` package brings the power of parallel processing to R. But before you want to use multi core CPUs you have to assign a multi core cluster. The `doSNOW` package is one possibility.

A simple use of the foreach loop is to calculate the sum of the square root and the square of all numbers from 1 to 100000.

```
library(foreach)
library(doSNOW)

cl <- makeCluster(5, type = "SOCK")
registerDoSNOW(cl)

f <- foreach(i = 1:100000, .combine = c, .inorder = F) %dopar% {
    k <- i ** 2 + sqrt(i)
    k
}
```

The structure of the output of `foreach` is controlled by the `.combine` argument. The default output structure is a list. In the code above, `c` is used to return a vector instead. Note that a calculation function (or operator) such as `"+"` may also be used to perform a calculation and return a further processed object.

It is important to mention that the result of each foreach-loop is the last call. Thus, in this example `k` will be added to the result.

Parameter	Details
.combine	combine Function. Determines how the results of the loop are combined. Possible values are `c`, `cbind`, `rbind`, `"+"`, `"*"`...
.inorder	if `TRUE` the result is ordered according to the order of the iteration vairable (here `i`). If `FALSE` the result is not ordered. This can have postive effects on computation time.

Parameter	Details
.packages	for functions which are provided by any package except `base`, like e.g. `mass`, `randomForest` or else, you have to provide these packages with `c("mass", "randomForest")`

Parallel processing with parallel package

The base package `parallel` allows parallel computation through forking, sockets, and random-number generation.

Detect the number of cores present on the localhost:

```
parallel::detectCores(all.tests = FALSE, logical = TRUE)
```

Create a cluster of the cores on the localhost:

```
parallelCluster <- parallel::makeCluster(parallel::detectCores())
```

First, a function appropriate for parallelization must be created. Consider the `mtcars` dataset. A regression on `mpg` could be improved by creating a separate regression model for each level of `cyl`.

```
data <- mtcars
yfactor <- 'cyl'
zlevels <- sort(unique(data[[yfactor]]))
datay <- data[,1]
dataz <- data[,2]
datax <- data[,3:11]


fitmodel <- function(zlevel, datax, datay, dataz) {
  glm.fit(x = datax[dataz == zlevel,], y = datay[dataz == zlevel])
}
```

Create a function that can loop through all the possible iterations of `zlevels`. This is still in serial, but is an important step as it determines the exact process that will be parallelized.

```
fitmodel <- function(zlevel, datax, datay, dataz) {
  glm.fit(x = datax[dataz == zlevel,], y = datay[dataz == zlevel])
}


for (zlevel in zlevels) {
  print("*****")
  print(zlevel)
  print(fitmodel(zlevel, datax, datay, dataz))
}
```

Curry this function:

```
worker <- function(zlevel) {
    fitmodel(zlevel,datax, datay, dataz)
```


www.dbooks.org

```
  }
```

Parallel computing using `parallel` cannot access the global environment. Luckily, each function creates a local environment `parallel` can access. Creation of a wrapper function allows for parallelization. The function to be applied also needs to be placed within the environment.

```
wrapper <- function(datax, datay, dataz) {
  # force evaluation of all paramters not supplied by parallelization apply
  force(datax)
  force(datay)
  force(dataz)
  # these variables are now in an enviroment accessible by parallel function

  # function to be applied also in the environment
  fitmodel <- function(zlevel, datax, datay, dataz) {
    glm.fit(x = datax[dataz == zlevel,], y = datay[dataz == zlevel])
  }

  # calling in this environment iterating over single parameter zlevel
  worker <- function(zlevel) {
    fitmodel(zlevel,datax, datay, dataz)
  }
  return(worker)
}
```

Now create a cluster and run the wrapper function.

```
parallelcluster <- parallel::makeCluster(parallel::detectCores())
models <- parallel::parLapply(parallelcluster,zlevels,
                              wrapper(datax, datay, dataz))
```

Always stop the cluster when finished.

```
parallel::stopCluster(parallelcluster)
```

The `parallel` package includes the entire `apply()` family, prefixed with `par`.

Random Number Generation

A major problem with parallelization is the used of RNG as seeds. Random numbers by the number are iterated by the number of operations from either the start of the session or the most recent `set.seed()`. Since parallel processes arise from the same function, it can use the same seed, possibly causing identical results! Calls will run in serial on the different cores, provide no advantage.

A set of seeds must be generated and sent to each parallel process. This is automatically done in some packages (`parallel`, `snow`, etc.), but must be explicitly addressed in others.

```
s <- seed
for (i in 1:numofcores) {
    s <- nextRNGStream(s)
    # send s to worker i as .Random.seed
```

```
}
```

Seeds can be also be set for reproducibility.

```
clusterSetRNGStream(cl = parallelcluster, iseed)
```

mcparallelDo

The `mcparallelDo` package allows for the evaluation of R code asynchronously on Unix-alike (e.g. Linux and MacOSX) operating systems. The underlying philosophy of the package is aligned with the needs of exploratory data analysis rather than coding. For coding asynchrony, consider the `future` package.

Example

Create data

```
data(ToothGrowth)
```

Trigger mcparallelDo to perform analysis on a fork

```
mcparallelDo({glm(len ~ supp * dose, data=ToothGrowth)},"interactionPredictorModel")
```

Do other things, e.g.

```
binaryPredictorModel <- glm(len ~ supp, data=ToothGrowth)
gaussianPredictorModel <- glm(len ~ dose, data=ToothGrowth)
```

The result from mcparallelDo returns in your targetEnvironment, e.g. .GlobalEnv, when it is complete with a message (by default)

```
summary(interactionPredictorModel)
```

Other Examples

```
# Example of not returning a value until we return to the top level
for (i in 1:10) {
  if (i == 1) {
    mcparallelDo({2+2}, targetValue = "output")
  }
  if (exists("output")) print(i)
}

# Example of getting a value without returning to the top level
for (i in 1:10) {
  if (i == 1) {
```


www.dbooks.org

```
    mcparallelDo({2+2}, targetValue = "output")
  }
  mcparallelDoCheck()
  if (exists("output")) print(i)
}
```

Read Parallel processing online: https://riptutorial.com/r/topic/1677/parallel-processing

Chapter 75: Pattern Matching and Replacement

Introduction

This topic covers matching string patterns, as well as extracting or replacing them. For details on defining complicated patterns see Regular Expressions.

Syntax

- grep("query", "subject", optional_args)
- grepl("query", "subject", optional_args)
- gsub("(group1)(group2)", "\\group#", "subject")

Remarks

Differences from other languages

Escaped regex symbols (like `\1`) are must be escaped a second time (like `\\1`), not only in the `pattern` argument, but also in the `replacement` to `sub` and `gsub`.

By default, the pattern for all commands (grep, sub, regexpr) is not Perl Compatible Regular Expression (PCRE) so some things like lookarounds are not supported. However, each function accepts a `perl=TRUE` argument to enable them. See the R Regular Expressions topic for details.

Specialized packages

- stringi
- stringr

Examples

Making substitutions

```
# example data
test_sentences <- c("The quick brown fox quickly", "jumps over the lazy dog")
```

Let's make the brown fox red:

www.dbooks.org

```
sub("brown","red", test_sentences)
#[1] "The quick red fox quickly"       "jumps over the lazy dog"
```

Now, let's make the `"fast"` fox act `"fastly"`. This won't do it:

```
sub("quick", "fast", test_sentences)
#[1] "The fast red fox quickly"       "jumps over the lazy dog"
```

`sub` only makes the first available replacement, we need `gsub` for global replacement:

```
gsub("quick", "fast", test_sentences)
#[1] "The fast red fox fastly"       "jumps over the lazy dog"
```

See Modifying strings by substitution for more examples.

Finding Matches

```
# example data
test_sentences <- c("The quick brown fox", "jumps over the lazy dog")
```

Is there a match?

`grepl()` is used to check whether a word or regular expression exists in a string or character vector. The function returns a TRUE/FALSE (or "Boolean") vector.

Notice that we can check each string for the word "fox" and receive a Boolean vector in return.

```
grepl("fox", test_sentences)
#[1]  TRUE FALSE
```

Match locations

`grep` takes in a character string and a regular expression. It returns a numeric vector of indexes.This will return which sentence contains the word "fox" in it.

```
grep("fox", test_sentences)
#[1] 1
```

Matched values

To select sentences that match a pattern:

```
# each of the following lines does the job:
test_sentences[grep("fox", test_sentences)]
```

```
test_sentences[grepl("fox", test_sentences)]
grep("fox", test_sentences, value = TRUE)
# [1] "The quick brown fox"
```

Details

Since the `"fox"` pattern is just a word, rather than a regular expression, we could improve performance (with either `grep` or `grepl`) by specifying `fixed = TRUE`.

```
grep("fox", test_sentences, fixed = TRUE)
#[1] 1
```

To select sentences that *don't* match a pattern, one can use `grep` with `invert = TRUE`; or follow subsetting rules with `-grep(...)` or `!grepl(...)`.

In both `grepl(pattern, x)` and `grep(pattern, x)`, the `x` parameter is vectorized, the `pattern` parameter is not. As a result, you cannot use these directly to match `pattern[1]` against `x[1]`, `pattern[2]` against `x[2]`, and so on.

Summary of matches

After performing the e.g. the `grepl` command, maybe you want to get an overview about how many matches where `TRUE` or `FALSE`. This is useful e.g. in case of big data sets. In order to do so run the `summary` command:

```
# example data
test_sentences <- c("The quick brown fox", "jumps over the lazy dog")

# find matches
matches <- grepl("fox", test_sentences)

# overview
summary(matches)
```

Single and Global match.

When working with regular expressions one modifier for PCRE is `g` for global match.

In R matching and replacement functions have two version: first match and global match:

- `sub(pattern,replacement,text)` will replace the first occurrence of pattern by replacement in text
- `gsub(pattern,replacement,text)` will do the same as sub but for each occurrence of pattern
- `regexpr(pattern,text)` will return the position of match for the first instance of pattern
- `gregexpr(pattern,text)` will return all matches.

www.dbooks.org

Some random data:

```
set.seed(123)
teststring <- paste0(sample(letters,20),collapse="")

# teststring
#[1] "htjuwakqxzpgrsbncvyo"
```

Let's see how this works if we want to replace vowels by something else:

```
sub("[aeiouy]"," ** HERE WAS A VOWEL** ",teststring)
#[1] "htj ** HERE WAS A VOWEL** wakqxzpgrsbncvyo"

gsub("[aeiouy]"," ** HERE WAS A VOWEL** ",teststring)
#[1] "htj ** HERE WAS A VOWEL** w ** HERE WAS A VOWEL** kqxzpgrsbncv ** HERE WAS A VOWEL**  **
HERE WAS A VOWEL** "
```

Now let's see how we can find a consonant immediately followed by one or more vowel:

```
regexpr("[^aeiou][aeiou]+",teststring)
#[1] 3
#attr(,"match.length")
#[1] 2
#attr(,"useBytes")
#[1] TRUE
```

We have a match on position 3 of the string of length 2, i.e: `ju`

Now if we want to get all matches:

```
gregexpr("[^aeiou][aeiou]+",teststring)
#[[1]]
#[1]  3  5 19
#attr(,"match.length")
#[1] 2 2 2
#attr(,"useBytes")
#[1] TRUE
```

All this is really great, but this only give use positions of match and that's not so easy to get what is matched, and here comes `regmatches` it's sole purpose is to extract the string matched from regexpr, but it has a different syntax.

Let's save our matches in a variable and then extract them from original string:

```
matches <- gregexpr("[^aeiou][aeiou]+",teststring)
regmatches(teststring,matches)
#[[1]]
#[1] "ju" "wa" "yo"
```

This may sound strange to not have a shortcut, but this allow extraction from another string by the matches of our first one (think comparing two long vector where you know there's is a common pattern for the first but not for the second, this allow an easy comparison):

```
teststring2 <- "this is another string to match against"
regmatches(teststring2,matches)
#[[1]]
#[1] "is" " i" "ri"
```

Attention note: by default the pattern is not Perl Compatible Regular Expression, some things like lookarounds are not supported, but each function presented here allow for `perl=TRUE` argument to enable them.

Find matches in big data sets

In case of big data sets, the call of `grepl("fox", test_sentences)` does not perform well. Big data sets are e.g. crawled websites or million of Tweets, etc.

The first acceleration is the usage of the `perl = TRUE` option. Even faster is the option `fixed = TRUE`. A complete example would be:

```
# example data
test_sentences <- c("The quick brown fox", "jumps over the lazy dog")

grepl("fox", test_sentences, perl = TRUE)
#[1]  TRUE FALSE
```

In case of text mining, often a corpus gets used. A corpus cannot be used directly with `grepl`. Therefore, consider this function:

```
searchCorpus <- function(corpus, pattern) {
  return(tm_index(corpus, FUN = function(x) {
    grepl(pattern, x, ignore.case = TRUE, perl = TRUE)
  }))
}
```

Read Pattern Matching and Replacement online: https://riptutorial.com/r/topic/1123/pattern-matching-and-replacement

www.dbooks.org

Chapter 76: Performing a Permutation Test

Examples

A fairly general function

We will use the built in tooth growth dataset. We are interested in whether there is a statistically significant difference in tooth growth when the guinea pigs are given vitamin C vs orange juice.

Here's the full example:

```
teethVC = ToothGrowth[ToothGrowth$supp == 'VC',]
teethOJ = ToothGrowth[ToothGrowth$supp == 'OJ',]

permutationTest = function(vectorA, vectorB, testStat){
  N = 10^5
  fullSet = c(vectorA, vectorB)
  lengthA = length(vectorA)
  lengthB = length(vectorB)
  trials <- replicate(N,
                      {index <- sample(lengthB + lengthA, size = lengthA, replace = FALSE)
                      testStat((fullSet[index]), fullSet[-index])  } )
  trials
}
vec1 =teethVC$len;
vec2 =teethOJ$len;
subtractMeans = function(a, b){ return (mean(a) - mean(b))}
result = permutationTest(vec1, vec2, subtractMeans)
observedMeanDifference = subtractMeans(vec1, vec2)
result = c(result, observedMeanDifference)
hist(result)
abline(v=observedMeanDifference, col = "blue")
pValue = 2*mean(result <= (observedMeanDifference))
pValue
```

After we read in the CSV, we define the function

```
permutationTest = function(vectorA, vectorB, testStat){
  N = 10^5
  fullSet = c(vectorA, vectorB)
  lengthA = length(vectorA)
  lengthB = length(vectorB)
  trials <- replicate(N,
                      {index <- sample(lengthB + lengthA, size = lengthA, replace = FALSE)
                      testStat((fullSet[index]), fullSet[-index])  } )
  trials
}
```

This function takes two vectors, and shuffles their contents together, then performs the function `testStat` on the shuffled vectors. The result of `teststat` is added to `trials`, which is the return value.

It does this `N = 10^5` times. Note that the value `N` could very well have been a parameter to the

function.

This leaves us with a new set of data, `trials`, the set of means that might result if there truly is no relationship between the two variables.

Now to define our test statistic:

```
subtractMeans = function(a, b){ return (mean(a) - mean(b))}
```

Perform the test:

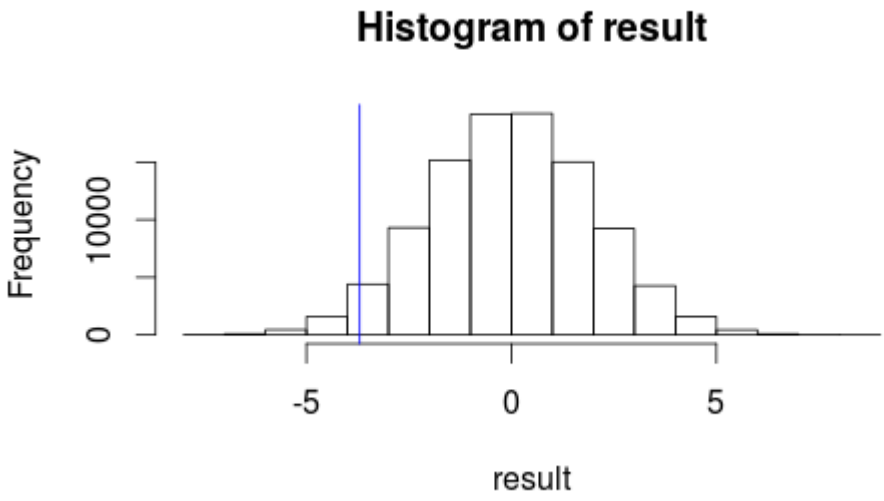
```
result = permutationTest(vec1, vec2, subtractMeans)
```

Calculate our actual observed mean difference:

```
observedMeanDifference = subtractMeans(vec1, vec2)
```

Let's see what our observation looks like on a histogram of our test statistic.

```
hist(result)
abline(v=observedMeanDifference, col = "blue")
```



It doesn't *look* like our observed result is very likely to occur by random chance...

We want to calculate the p-value, the likeliehood of the original observed result if their is no relationship between the two variables.

```
pValue = 2*mean(result >= (observedMeanDifference))
```

Let's break that down a bit:

```
result >= (observedMeanDifference)
```


www.dbooks.org

Will create a boolean vector, like:

```
FALSE TRUE FALSE FALSE TRUE FALSE ...
```

With `TRUE` every time the value of `result` is greater than or equal to the `observedMean`.

The function `mean` will interpret this vector as `1` for `TRUE` and `0` for `FALSE`, and give us the percentage of `1`'s in the mix, ie the number of times our shuffled vector mean difference surpassed or equalled what we observed.

Finally, we multiply by 2 because the distribution of our test statistic is highly symmetric, and we really want to know which results are "more extreme" than our observed result.

All that's left is to output the p-value, which turns out to be `0.06093939`. Interpretation of this value is subjective, but I would say that it looks like Vitamin C promotes tooth growth quite a lot more than Orange Juice does.

Read Performing a Permutation Test online: https://riptutorial.com/r/topic/3216/performing-a-permutation-test

Chapter 77: Pipe operators (%>% and others)

Introduction

Pipe operators, available in `magrittr`, `dplyr`, and other R packages, process a data-object using a sequence of operations by passing the result of one step as input for the next step using infix-operators rather than the more typical R method of nested function calls.

Note that the intended aim of pipe operators is to increase human readability of written code. See Remarks section for performance considerations.

Syntax

- lhs %>% rhs # pipe syntax for `rhs(lhs)`
- lhs %>% rhs(a = 1) # pipe syntax for `rhs(lhs, a = 1)`
- lhs %>% rhs(a = 1, b = .) # pipe syntax for `rhs(a = 1, b = lhs)`
- lhs %<>% rhs # pipe syntax for `lhs <- rhs(lhs)`
- lhs %$% rhs(a) # pipe syntax for `with(lhs, rhs(lhs$a))`
- lhs %T>% rhs # pipe syntax for `{ rhs(lhs); lhs }`

Parameters

lhs	rhs
A value or the magrittr placeholder.	A function call using the magrittr semantics

Remarks

Packages that use `%>%`

The pipe operator is defined in the `magrittr` package, but it gained huge visibility and popularity with the `dplyr` package (which imports the definition from `magrittr`). Now it is part of `tidyverse`, which is a collection of packages that *"work in harmony because they share common data representations and API design"*.

The `magrittr` package also provides several variations of the pipe operator for those who want more flexibility in piping, such as the compound assignment pipe `%<>%`, the exposition pipe `%$%`, and the tee operator `%T>%`. It also provides a suite of alias functions to replace common functions that have special syntax (`+`, `[`, `[[`, etc.) so that they can be easily used within a chain of pipes.

www.dbooks.org

Finding documentation

As with any *infix operator* (such as `+`, `*`, `^`, `&`, `%in%`), you can find the official documentation if you put it in quotes: `?'%>%'` or `help('%>%')` (assuming you have loaded a package that attaches `pkg:magrittr`).

Hotkeys

There is a special hotkey in RStudio for the pipe operator: `Ctrl+Shift+M` (*Windows & Linux*), `Cmd+Shift+M` (*Mac*).

Performance Considerations

While the pipe operator is useful, be aware that there is a negative impact on performance due mainly to the overhead of using it. Consider the following two things carefully when using the pipe operator:

- Machine performance (loops)
- Evaluation (`object %>% rm()` does not remove `object`)

Examples

Basic use and chaining

The pipe operator, `%>%`, is used to insert an argument into a function. It is not a base feature of the language and can only be used after attaching a package that provides it, such as `magrittr`. The pipe operator takes the left-hand side (LHS) of the pipe and uses it as the first argument of the function on the right-hand side (RHS) of the pipe. For example:

```
library(magrittr)

1:10 %>% mean
# [1] 5.5

# is equivalent to
mean(1:10)
# [1] 5.5
```

The pipe can be used to replace a sequence of function calls. Multiple pipes allow us to read and write the sequence from left to right, rather than from inside to out. For example, suppose we have `years` defined as a factor but want to convert it to a numeric. To prevent possible information loss, we first convert to character and then to numeric:

```
years <- factor(2008:2012)

# nesting
as.numeric(as.character(years))

# piping
```

```
years %>% as.character %>% as.numeric
```

If we don't want the LHS (Left Hand Side) used as the *first* argument on the RHS (Right Hand Side), there are workarounds, such as naming the arguments or using `.` to indicate where the piped input goes.

```
# example with grepl
# its syntax:
# grepl(pattern, x, ignore.case = FALSE, perl = FALSE, fixed = FALSE, useBytes = FALSE)

# note that the `substring` result is the *2nd* argument of grepl
grepl("Wo", substring("Hello World", 7, 11))

# piping while naming other arguments
"Hello World" %>% substring(7, 11) %>% grepl(pattern = "Wo")

# piping with .
"Hello World" %>% substring(7, 11) %>% grepl("Wo", .)

# piping with . and curly braces
"Hello World" %>% substring(7, 11) %>% { c(paste('Hi', .)) }
#[1] "Hi World"

#using LHS multiple times in argument with curly braces and .
"Hello World" %>% substring(7, 11) %>% { c(paste(. ,'Hi', .)) }
#[1] "World Hi World"
```

Functional sequences

Given a sequence of steps we use repeatedly, it's often handy to store it in a function. Pipes allow for saving such functions in a readable format by starting a sequence with a dot as in:

```
. %>% RHS
```

As an example, suppose we have factor dates and want to extract the year:

```
library(magrittr) # needed to include the pipe operators
library(lubridate)
read_year <- . %>% as.character %>% as.Date %>% year

# Creating a dataset
df <- data.frame(now = "2015-11-11", before = "2012-01-01")
#          now     before
# 1 2015-11-11 2012-01-01

# Example 1: applying `read_year` to a single character-vector
df$now %>% read_year
# [1] 2015

# Example 2: applying `read_year` to all columns of `df`
df %>% lapply(read_year) %>% as.data.frame  # implicit `lapply(df, read_year)
#    now before
# 1 2015   2012

# Example 3: same as above using `mutate_all`
```


www.dbooks.org

```
library(dplyr)
df %>% mutate_all(funs(read_year))
# if an older version of dplyr use `mutate_each`
#    now before
# 1 2015   2012
```

We can review the composition of the function by typing its name or using `functions`:

```
read_year
# Functional sequence with the following components:
#
#  1. as.character(.)
#  2. as.Date(.)
#  3. year(.)
#
# Use 'functions' to extract the individual functions.
```

We can also access each function by its position in the sequence:

```
read_year[[2]]
# function (.)
# as.Date(.)
```

Generally, this approach may be useful when clarity is more important than speed.

Assignment with %<>%

The `magrittr` package contains a compound assignment infix-operator, `%<>%`, that updates a value by first piping it into one or more `rhs` expressions and then assigning the result. This eliminates the need to type an object name twice (once on each side of the assignment operator `<-`). `%<>%` must be the first infix-operator in a chain:

```
library(magrittr)
library(dplyr)

df <- mtcars
```

Instead of writing

```
df <- df %>% select(1:3) %>% filter(mpg > 20, cyl == 6)
```

or

```
df %>% select(1:3) %>% filter(mpg > 20, cyl == 6) -> df
```

The compound assignment operator will both pipe and reassign `df`:

```
df %<>% select(1:3) %>% filter(mpg > 20, cyl == 6)
```

Exposing contents with %$%

The exposition pipe operator, `%$%`, exposes the column names as R symbols within the left-hand side object to the right-hand side expression. This operator is handy when piping into functions that do not have a `data` argument (unlike, say, `lm`) and that don't take a data.frame and column names as arguments (most of the main dplyr functions).

The exposition pipe operator `%$%` allows a user to avoid breaking a pipeline when needing to refer to column names. For instance, say you want to filter a data.frame and then run a correlation test on two columns with `cor.test`:

```
library(magrittr)
library(dplyr)
mtcars %>%
  filter(wt > 2) %$%
  cor.test(hp, mpg)

#>
#>  Pearson's product-moment correlation
#>
#> data:  hp and mpg
#> t = -5.9546, df = 26, p-value = 2.768e-06
#> alternative hypothesis: true correlation is not equal to 0
#> 95 percent confidence interval:
#>  -0.8825498 -0.5393217
#> sample estimates:
#>        cor
#> -0.7595673
```

Here the standard `%>%` pipe passes the data.frame through to `filter()`, while the `%$%` pipe exposes the column names to `cor.test()`.

The exposition pipe works like a pipe-able version of the base R `with()` functions, and the same left-hand side objects are accepted as inputs.

Using the pipe with dplyr and ggplot2

The `%>%` operator can also be used to pipe the dplyr output into ggplot. This creates a unified exploratory data analysis (EDA) pipeline that is easily customizable. This method is faster than doing the aggregations internally in ggplot and has the added benefit of avoiding unnecessary intermediate variables.

```
library(dplyr)
library(ggplot)


diamonds %>%
    filter(depth > 60) %>%
    group_by(cut) %>%
    summarize(mean_price = mean(price)) %>%
    ggplot(aes(x = cut, y = mean_price)) +
        geom_bar(stat = "identity")
```

Creating side effects with %T>%

www.dbooks.org

Some functions in R produce a side effect (i.e. saving, printing, plotting, etc) and do not always return a meaningful or desired value.

`%T>%` (tee operator) allows you to forward a value into a side-effect-producing function while keeping the original `lhs` value intact. In other words: the tee operator works like `%>%`, except the return values is `lhs` itself, and not the result of the `rhs` function/expression.

Example: Create, pipe, write, and return an object. If `%>%` were used in place of `%T>%` in this example, then the variable `all_letters` would contain `NULL` rather than the value of the sorted object.

```
all_letters <- c(letters, LETTERS) %>%
    sort %T>%
    write.csv(file = "all_letters.csv")

read.csv("all_letters.csv") %>% head()
#   x
# 1 a
# 2 A
# 3 b
# 4 B
# 5 c
# 6 C
```

Warning: Piping an unnamed object to `save()` will produce an object named `.` when loaded into the workspace with `load()`. However, a workaround using a helper function is possible (which can also be written inline as an anonymous function).

```
all_letters <- c(letters, LETTERS) %>%
    sort %T>%
    save(file = "all_letters.RData")

load("all_letters.RData", e <- new.env())

get("all_letters", envir = e)
# Error in get("all_letters", envir = e) : object 'all_letters' not found

get(".", envir = e)
#  [1] "a" "A" "b" "B" "c" "C" "d" "D" "e" "E" "f" "F" "g" "G" "h" "H" "i" "I" "j" "J"
# [21] "k" "K" "l" "L" "m" "M" "n" "N" "o" "O" "p" "P" "q" "Q" "r" "R" "s" "S" "t" "T"
# [41] "u" "U" "v" "V" "w" "W" "x" "X" "y" "Y" "z" "Z"

# Work-around
save2 <- function(. = ., name, file = stop("'file' must be specified")) {
  assign(name, .)
  call_save <- call("save", ... = name, file = file)
  eval(call_save)
}

all_letters <- c(letters, LETTERS) %>%
    sort %T>%
    save2("all_letters", "all_letters.RData")
```

Read Pipe operators (%>% and others) online: https://riptutorial.com/r/topic/652/pipe-operators-----and-others-

Chapter 78: Pivot and unpivot with data.table

Syntax

- Melt with `melt(DT, id.vars=c(..), variable.name="CategoryLabel", value.name="Value")`
- Cast with `dcast(DT, LHS ~ RHS, value.var="Value", fun.aggregate=sum)`

Parameters

Parameter	Details
id.vars	tell `melt` which columns to retain
variable.name	tell `melt` what to call the column with category labels
value.name	tell `melt` what to call the column that has values associated with category labels
value.var	tell `dcast` where to find the values to cast in columns
formula	tell `dcast` which columns to retain to form a unique record identifier (LHS) and which one holds the category labels (RHS)
fun.aggregate	specify the function to use when the casting operation generates a list of values in each cell

Remarks

Much of what goes into conditioning data to build models or visualizations can be accomplished with `data.table`. As compare to other options, `data.table` offers advantages of speed and flexibility.

Examples

Pivot and unpivot tabular data with data.table - I

Convert from wide form to long form

Load `data USArrests` from `datasets`.

```
data("USArrests")
head(USArrests)

           Murder Assault UrbanPop Rape
Alabama      13.2     236       58 21.2
Alaska       10.0     263       48 44.5
```


www.dbooks.org

```
Arizona       8.1     294       80 31.0
Arkansas      8.8     190       50 19.5
California    9.0     276       91 40.6
Colorado      7.9     204       78 38.7
```

Use `?USArrests` to find out more. First, convert to `data.table`. The names of states are row names in the original `data.frame`.

```
library(data.table)
DT <- as.data.table(USArrests, keep.rownames=TRUE)
```

This is data in the wide form. It has a column for each variable. The data can also be stored in long form without loss of information. The long form has one column that stores the variable names. Then, it has another column for the variable values. The long form of `USArrests` looks like so.

```
            State    Crime  Rate
  1:       Alabama   Murder  13.2
  2:        Alaska   Murder  10.0
  3:       Arizona   Murder   8.1
  4:      Arkansas   Murder   8.8
  5:    California   Murder   9.0
 ---
196:      Virginia     Rape  20.7
197:    Washington     Rape  26.2
198: West Virginia     Rape   9.3
199:     Wisconsin     Rape  10.8
200:       Wyoming     Rape  15.6
```

We use the `melt` function to switch from wide form to long form.

```
DTm <- melt(DT)
names(DTm) <- c("State", "Crime", "Rate")
```

By default, `melt` treats all columns with numeric data as variables with values. In `USArrests`, the variable `UrbanPop` represents the percentage urban population of a state. It is different from the other variabes, `Murder`, `Assault` and `Rape`, which are violent crimes reported per 100,000 people. Suppose we want to retain `UrbanPop` column. We achieve this by setting `id.vars` as follows.

```
DTmu <- melt(DT, id.vars=c("rn", "UrbanPop" ),
             variable.name='Crime', value.name = "Rate")
names(DTmu)[1] <- "State"
```

Note that we have specified the names of the column containing category names (Murder, Assault, etc.) with `variable.name` and the column containing the values with `value.name`. Our data looks like so.

```
            State UrbanPop  Crime Rate
  1:       Alabama       58 Murder 13.2
  2:        Alaska       48 Murder 10.0
  3:       Arizona       80 Murder  8.1
  4:      Arkansas       50 Murder  8.8
```

```
  5:    California       91 Murder  9.0
```

Generating summaries with with split-apply-combine style approach is a breeze. For example, to summarize violent crimes by state?

```
DTmu[, .(ViolentCrime = sum(Rate)), by=State]
```

This gives:

```
        State ViolentCrime
1:    Alabama        270.4
2:     Alaska        317.5
3:    Arizona        333.1
4:   Arkansas        218.3
5: California        325.6
6:   Colorado        250.6
```

Pivot and unpivot tabular data with data.table - II

Convert from long form to wide form

To recover data from the previous example, use `dcast` like so.

```
DTc <- dcast(DTmu, State + UrbanPop ~ Crime)
```

This gives the data in the original wide form.

```
             State UrbanPop Murder Assault Rape
 1:        Alabama       58   13.2     236 21.2
 2:         Alaska       48   10.0     263 44.5
 3:        Arizona       80    8.1     294 31.0
 4:       Arkansas       50    8.8     190 19.5
 5:     California       91    9.0     276 40.6
```

Here, the formula notation is used to specify the columns that form a unique record identifier (LHS) and the column containing category labels for new column names (RHS). Which column to use for the numeric values? By default, `dcast` uses the first column with numerical values left over when from the formula specification. To make explicit, use the parameter `value.var` with column name.

When the operation produces a list of values in each cell, `dcast` provides a `fun.aggregate` method to handle the situation. Say I am interested in states with similar urban population when investigating crime rates. I add a column `Decile` with computed information.

```
DTmu[, Decile := cut(UrbanPop, quantile(UrbanPop, probs = seq(0, 1, by=0.1)))]
levels(DTmu$Decile) <- paste0(1:10, "D")
```

Now, casting `Decile ~ Crime` produces multiple values per cell. I can use `fun.aggregate` to determine how these are handled. Both text and numerical values can be handle this way.

www.dbooks.org

```
dcast(DTmu, Decile ~ Crime, value.var="Rate", fun.aggregate=sum)
```

This gives:

```
dcast(DTmu, Decile ~ Crime, value.var="Rate", fun.aggregate=mean)
```

This gives:

```
            State UrbanPop  Crime Rate Decile
 1:       Alabama       58 Murder 13.2     4D
 2:        Alaska       48 Murder 10.0     2D
 3:       Arizona       80 Murder  8.1     8D
 4:      Arkansas       50 Murder  8.8     2D
 5:    California       91 Murder  9.0    10D
```

There are multiple states in each decile of the urban population. Use `fun.aggregate` to specify how these should be handled.

```
dcast(DTmu, Decile ~ Crime, value.var="Rate", fun.aggregate=sum)
```

This sums over the data for like states, giving the following.

```
   Decile Murder Assault  Rape
1:     1D   39.4     808  62.6
2:     2D   35.3     815  94.3
3:     3D   22.6     451  67.7
4:     4D   54.9     898 106.0
5:     5D   42.4     758 107.6
```

Read Pivot and unpivot with data.table online: https://riptutorial.com/r/topic/6934/pivot-and-unpivot-with-data-table

Chapter 79: Probability Distributions with R

Examples

PDF and PMF for different distributions in R

PMF FOR THE BINOMIAL DISTRIBUTION

Suppose that a fair die is rolled 10 times. What is the probability of throwing exactly two sixes?

You can answer the question using the dbinom function:

```
> dbinom(2, 10, 1/6)
[1] 0.29071
```

PMF FOR THE POISSON DISTRIBUTION

The number of sandwhich ordered in a restaurant on a given day is known to follow a Poisson distribution with a mean of 20. What is the probability that exactly eighteen sandwhich will be ordered tomorrow?

You can answer the question with the dpois function:

```
> dpois(18, 20)
[1] 0.08439355
```

PDF FOR THE NORMAL DISTRIBUTION

To find the value of the pdf at x=2.5 for a normal distribution with a mean of 5 and a standard deviation of 2, use the command:

```
> dnorm(2.5, mean=5, sd=2)
[1] 0.09132454
```

Read Probability Distributions with R online: https://riptutorial.com/r/topic/4333/probability-distributions-with-r

www.dbooks.org

Chapter 80: Publishing

Introduction

There are many ways of formatting R code, tables and graphs for publishing.

Remarks

R users often want to publish analysis and results in a reproducible way. See Reproducible R for details.

Examples

Formatting tables

Here, "table" is meant broadly (covering `data.frame`, `table`,

Printing to plain text

Printing (as seen in the console) might suffice for a plain-text document to be viewed in monospaced font:

Note: Before making the example data below, make sure you're in an empty folder you can write to. Run `getwd()` and read `?setwd` if you need to change folders.

```
..w = options()$width
options(width = 500) # reduce text wrapping
sink(file = "mytab.txt")
   summary(mtcars)
sink()
options(width = ..w)
rm(..w)
```

Printing delimited tables

Writing to CSV (or another common format) and then opening in a spreadsheet editor to apply finishing touches is another option:

Note: Before making the example data below, make sure you're in an empty folder you can write to. Run `getwd()` and read `?setwd` if you need to change folders.

```
write.csv(mtcars, file="mytab.csv")
```

Further resources

- `knitr::kable`
- stargazer
- `tables::tabular`
- texreg
- xtable

Formatting entire documents

`Sweave` from the `utils` package allows for formatting code, prose, graphs and tables together in a LaTeX document.

Further Resources

- Knitr and RMarkdown

Read Publishing online: https://riptutorial.com/r/topic/9039/publishing

www.dbooks.org

Chapter 81: R code vectorization best practices

Examples

By row operations

The key in vectorizing R code, is to reduce or eliminate "by row operations" or method dispatching of R functions.

That means that when approaching a problem that at first glance requires "by row operations", such as calculating the means of each row, one needs to ask themselves:

- What are the classes of the data sets I'm dealing with?
- Is there an existing compiled code that can achieve this without the need of repetitive evaluation of R functions?
- If not, can I do these operation by columns instead by row?
- Finally, is it worth spending a lot of time on developing complicated vectorized code instead of just running a simple `apply` loop? In other words, is the data big/sophisticated enough that R can't handle it efficiently using a simple loop?

Putting aside the memory pre-allocation issue and growing object in loops, we will focus in this example on how to possibly avoid `apply` loops, method dispatching or re-evaluating R functions within loops.

A standard/easy way of calculating mean by row would be:

```
apply(mtcars, 1, mean)
          Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive   Hornet
Sportabout             Valiant          Duster 360
           29.90727            29.98136            23.59818            38.73955
53.66455            35.04909            59.72000
          Merc 240D            Merc 230            Merc 280           Merc 280C          Merc
450SE          Merc 450SL         Merc 450SLC
           24.63455            27.23364            31.86000            31.78727
46.43091            46.50000            46.35000
 Cadillac Fleetwood Lincoln Continental   Chrysler Imperial            Fiat 128         Honda
Civic      Toyota Corolla       Toyota Corona
           66.23273            66.05855            65.97227            19.44091
17.74227            18.81409            24.88864
   Dodge Challenger         AMC Javelin          Camaro Z28    Pontiac Firebird           Fiat
X1-9       Porsche 914-2        Lotus Europa
           47.24091            46.00773            58.75273            57.37955
18.92864            24.77909            24.88027
     Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
           60.97182            34.50818            63.15545            26.26273
```

But can we do better? Lets's see what happened here:

1. First, we converted a `data.frame` to a `matrix`. (Note that his happens within the `apply` function.) This is both inefficient and dangerous. a `matrix` can't hold several column types at a time. Hence, such conversion will probably lead to loss of information and some times to misleading results (compare `apply(iris, 2, class)` with `str(iris)` or with `sapply(iris, class)`).
2. Second of all, we performed an operation repetitively, one time for each row. Meaning, we had to evaluate some R function `nrow(mtcars)` times. In this specific case, `mean` is not a computationally expensive function, hence R could likely easily handle it even for a big data set, but what would happen if we need to calculate the standard deviation by row (which involves an expensive square root operation)? Which brings us to the next point:
3. We evaluated the R function many times, but maybe there already is a compiled version of this operation?

Indeed we could simply do:

```
rowMeans(mtcars)
          Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive   Hornet
Sportabout             Valiant          Duster 360
           29.90727            29.98136            23.59818            38.73955
53.66455            35.04909            59.72000
          Merc 240D            Merc 230            Merc 280           Merc 280C          Merc
450SE           Merc 450SL         Merc 450SLC
           24.63455            27.23364            31.86000            31.78727
46.43091            46.50000            46.35000
 Cadillac Fleetwood Lincoln Continental   Chrysler Imperial            Fiat 128         Honda
Civic      Toyota Corolla       Toyota Corona
           66.23273            66.05855            65.97227            19.44091
17.74227            18.81409            24.88864
   Dodge Challenger         AMC Javelin          Camaro Z28    Pontiac Firebird           Fiat
X1-9       Porsche 914-2        Lotus Europa
           47.24091            46.00773            58.75273            57.37955
18.92864            24.77909            24.88027
     Ford Pantera L        Ferrari Dino       Maserati Bora          Volvo 142E
           60.97182            34.50818            63.15545            26.26273
```

This involves no by row operations and therefore no repetitive evaluation of R functions. **However**, we still converted a `data.frame` to a `matrix`. Though `rowMeans` has an error handling mechanism and it won't run on a data set that it can't handle, it's still has an efficiency cost.

```
rowMeans(iris)
Error in rowMeans(iris) : 'x' must be numeric
```

But still, can we do better? We could try instead of a matrix conversion with error handling, a different method that will allow us to use `mtcars` as a vector (because a `data.frame` is essentially a `list` and a `list` is a `vector`).

```
Reduce(`+`, mtcars)/ncol(mtcars)
 [1] 29.90727 29.98136 23.59818 38.73955 53.66455 35.04909 59.72000 24.63455 27.23364 31.86000
31.78727 46.43091 46.50000 46.35000 66.23273 66.05855
[17] 65.97227 19.44091 17.74227 18.81409 24.88864 47.24091 46.00773 58.75273 57.37955 18.92864
24.77909 24.88027 60.97182 34.50818 63.15545 26.26273
```


www.dbooks.org

Now for possible speed gain, we lost column names and error handling (including `NA` handling).

Another example would be calculating mean by group, using base R we could try

```
aggregate(. ~ cyl, mtcars, mean)
cyl      mpg     disp        hp     drat       wt     qsec        vs        am     gear
carb
1   4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090909
1.545455
2   6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857143
3.428571
3   8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285714
3.500000
```

Still, we are basically evaluating an R function in a loop, but the loop is now hidden in an internal C function (it matters little whether it is a C or an R loop).

Could we avoid it? Well there is a compiled function in R called `rowsum`, hence we could do:

```
rowsum(mtcars[-2], mtcars$cyl)/table(mtcars$cyl)
mpg     disp        hp     drat       wt     qsec        vs        am     gear     carb
4 26.66364 105.1364  82.63636 4.070909 2.285727 19.13727 0.9090909 0.7272727 4.090909 1.545455
6 19.74286 183.3143 122.28571 3.585714 3.117143 17.97714 0.5714286 0.4285714 3.857143 3.428571
8 15.10000 353.1000 209.21429 3.229286 3.999214 16.77214 0.0000000 0.1428571 3.285714 3.500000
```

Though we had to convert to a matrix first too.

A this point we may question whether our current data structure is the most appropriate one. Is a `data.frame` is the best practice? Or should one just switch to a `matrix` data structure in order to gain efficiency?

By row operations will get more and more expensive (even in matrices) as we start to evaluate expensive functions each time. Lets us consider a variance calculation by row example.

Lets say we have a matrix `m`:

```
set.seed(100)
m <- matrix(sample(1e2), 10)
m
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    8   33   39   86   71  100   81   68   89    84
 [2,]   12   16   57   80   32   82   69   11   41    92
 [3,]   62   91   53   13   42   31   60   70   98    79
 [4,]   66   94   29   67   45   59   20   96   64     1
 [5,]   36   63   76    6   10   48   85   75   99     2
 [6,]   18    4   27   19   44   56   37   95   26    40
 [7,]    3   24   21   25   52   51   83   28   49    17
 [8,]   46    5   22   43   47   74   35   97   77    65
 [9,]   55   54   78   34   50   90   30   61   14    58
[10,]   88   73   38   15    9   72    7   93   23    87
```

One could simply do:

```
apply(m, 1, var)
[1]  871.6556  957.5111  699.2111  941.4333 1237.3333  641.8222  539.7889  759.4333  500.4889
1255.6111
```

On the other hand, one could also completely vectorize this operation by following the formula of variance

```
RowVar <- function(x) {
  rowSums((x - rowMeans(x))^2)/(dim(x)[2] - 1)
}
RowVar(m)
[1]  871.6556  957.5111  699.2111  941.4333 1237.3333  641.8222  539.7889  759.4333  500.4889
1255.6111
```

Read R code vectorization best practices online: https://riptutorial.com/r/topic/3327/r-code-vectorization-best-practices

www.dbooks.org

Chapter 82: R in LaTeX with knitr

Syntax

1. <<internal-code-chunk-name, options...>>=
   # R Code Here
   @
2. \Sexpr{ #R Code Here }
3. << read-external-R-file >>=
   read_chunk('r-file.R')
   @
   <<external-code-chunk-name, options...>>=
   @

Parameters

Option	Details
echo	(TRUE/FALSE) - whether to include R source code in the output file
message	(TRUE/FALSE) - whether to include messages from the R source execution in the output file
warning	(TRUE/FALSE) - whether to include warnings from the R source execution in the output file
error	(TRUE/FALSE) - whether to include errors from the R source execution in the output file
cache	(TRUE/FALSE) - whether to cache the results of the R source execution
fig.width	(numeric) - width of the plot generated by the R source execution
fig.height	(numeric) - height of the plot generated by the R source execution

Remarks

Knitr is a tool that allows us to interweave natural language (in the form of LaTeX) and source code (in the form of R). In general, the concept of interspersing natural language and source code is called literate programming. Since knitr files contain a mixture of LaTeX (traditionally housed in .tex files) and R (traditionally housed in .R files) a new file extension called R noweb (.Rnw) is required. .Rnw files contain a mixture of LaTeX and R code.

Knitr allows for the generation of statistical reports in PDF format and is a key tool for achieving reproducable research.

Compiling .Rnw files to a PDF is a two step process. First, we need to know how to execute the R code and capture the output in a format that a LaTeX compiler can understand (a process called 'kniting'). We do this using the knitr package. The command for this is shown below, assuming you have installed the knitr package:

```
Rscript -e "library(knitr); knit('r-noweb-file.Rnw')
```

This will generate a normal .tex file (called r-noweb.tex in this example) which can then be turned into a PDF file using:

```
pdflatex r-noweb-file.tex
```

Examples

R in Latex with Knitr and Code Externalization

Knitr is an R package that allows us to intermingle R code with LaTeX code. One way to achieve this is external code chunks. External code chunks allow us to develop/test R Scripts in an R development environment and then include the results in a report. It is a powerful organizational technique. This approach is demonstrated below.

```
# r-noweb-file.Rnw
\documentclass{article}

 <<echo=FALSE,cache=FALSE>>=
 knitr::opts_chunk$set(echo=FALSE,  cache=TRUE)
 knitr::read_chunk('r-file.R')
 @

\begin{document}
This is an Rnw file (R noweb).  It contains a combination of LateX and R.

One we have called the read\_chunk command above we can reference sections of code in the r-
file.R script.

<<Chunk1>>=
@
\end{document}
```

When using this approach we keep our code in a separate R file as shown below.

```
## r-file.R
## note the specific comment style of a single pound sign followed by four dashes

# ---- Chunk1 ----

print("This is R Code in an external file")

x <- seq(1:10)
y <- rev(seq(1:10))
plot(x,y)
```


www.dbooks.org

R in Latex with Knitr and Inline Code Chunks

Knitr is an R package that allows us to intermingle R code with LaTeX code. One way to achieve this is inline code chunks. This apporach is demonstrated below.

```
# r-noweb-file.Rnw
\documentclass{article}
\begin{document}
This is an Rnw file (R noweb).  It contains a combination of LateX and R.

<<my-label>>=
print("This is an R Code Chunk")
x <- seq(1:10)
@

Above is an internal code chunk.
We can access data created in any code chunk inline with our LaTeX code like this.
The length of array x is \Sexpr{length(x)}.

\end{document}
```

R in LaTex with Knitr and Internal Code Chunks

Knitr is an R package that allows us to intermingle R code with LaTeX code. One way to achieve this is internal code chunks. This apporach is demonstrated below.

```
# r-noweb-file.Rnw
\documentclass{article}
\begin{document}
This is an Rnw file (R noweb).  It contains a combination of LateX and R.

<<code-chunk-label>>=
print("This is an R Code Chunk")
x <- seq(1:10)
y <- seq(1:10)
plot(x,y)  # Brownian motion
@

\end{document}
```

Read R in LaTeX with knitr online: https://riptutorial.com/r/topic/4334/r-in-latex-with-knitr

Chapter 83: R Markdown Notebooks (from RStudio)

Introduction

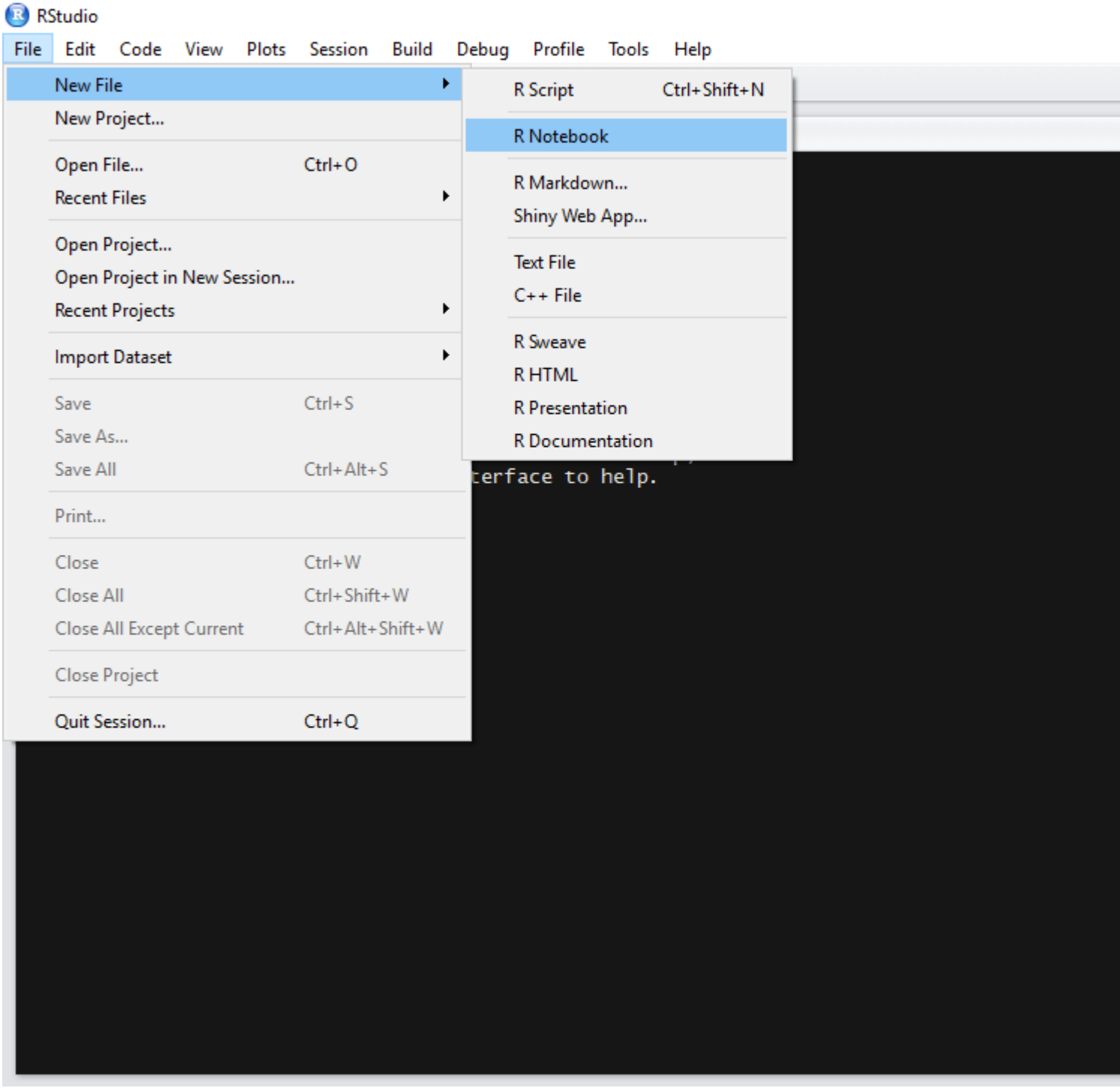
An R Notebook is an R Markdown document with chunks that can be executed independently and interactively, with output visible immediately beneath the input. They are similar to R Markdown documents with the exception of results being displayed in the R Notebook creation/edit mode rather than in the rendered output. **Note:** R Notebooks are new feature of RStudio and are only available in version 1.0 or higher of RStudio.

Examples

Creating a Notebook

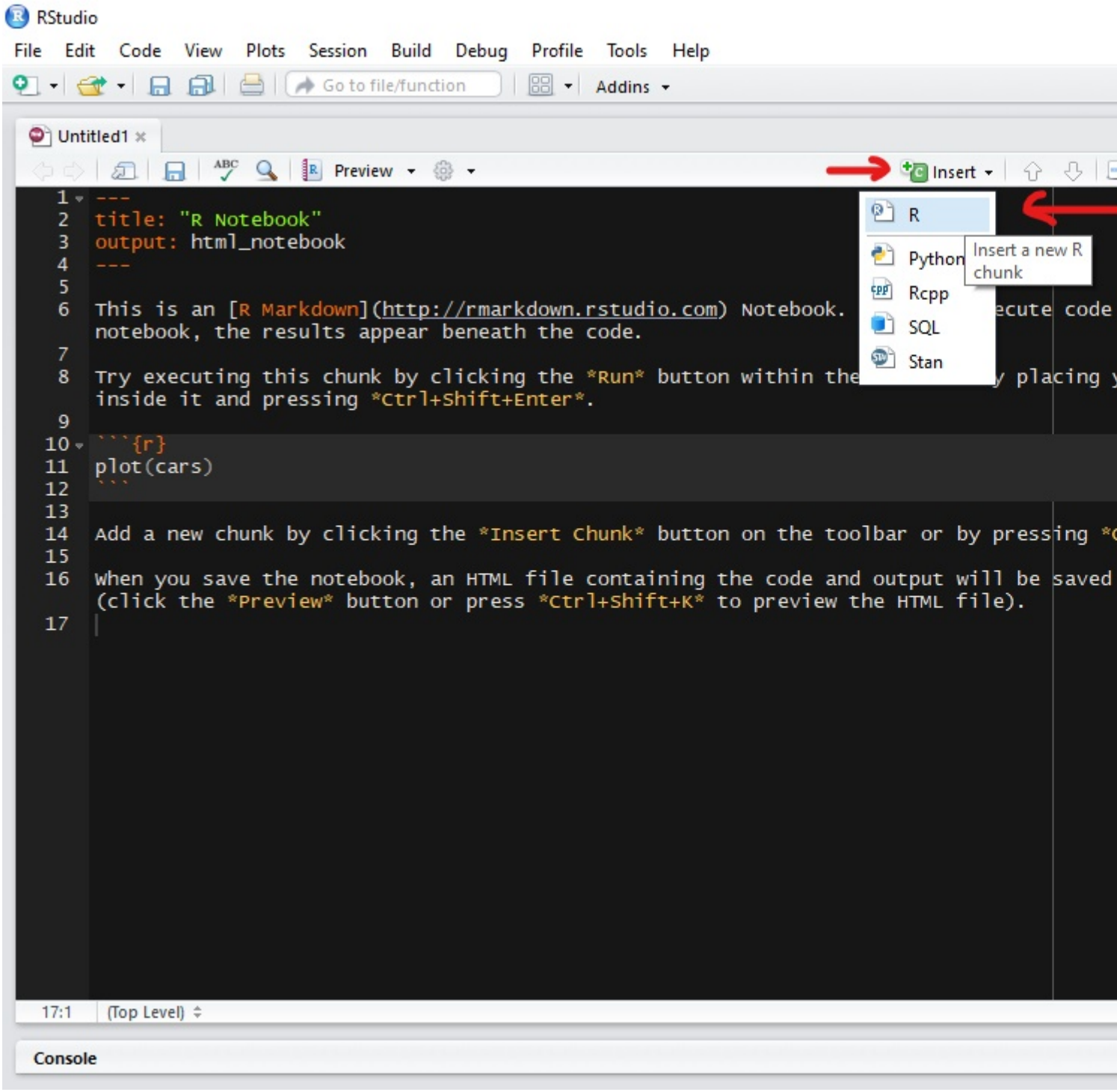
You can create a new notebook in RStudio with the menu command File -> New File -> R Notebook
If you don't see the option for R Notebook, then you need to update your version of RStudio. For installation of RStudio follow this guide

www.dbooks.org



Inserting Chunks

Chunks are pieces of code that can be executed interactively. In-order to insert a new chunk by clicking on the **insert** button present on the notebook toolbar and select your desired code platform (R in this case, since we want to write R code). Alternatively we can use keyboard shortcuts to insert a new chunk **Ctrl + Alt + I (OS X: Cmd + Option + I)**



Executing Chunk Code

You can run the current chunk by clicking **Run current Chunk (green play button)** present on the right side of the chunk. Alternatively we can use keyboard shortcut **Ctrl + Shift + Enter (OS X: Cmd + Shift + Enter)**

The output from all the lines in the chunk will appear beneath the chunk.

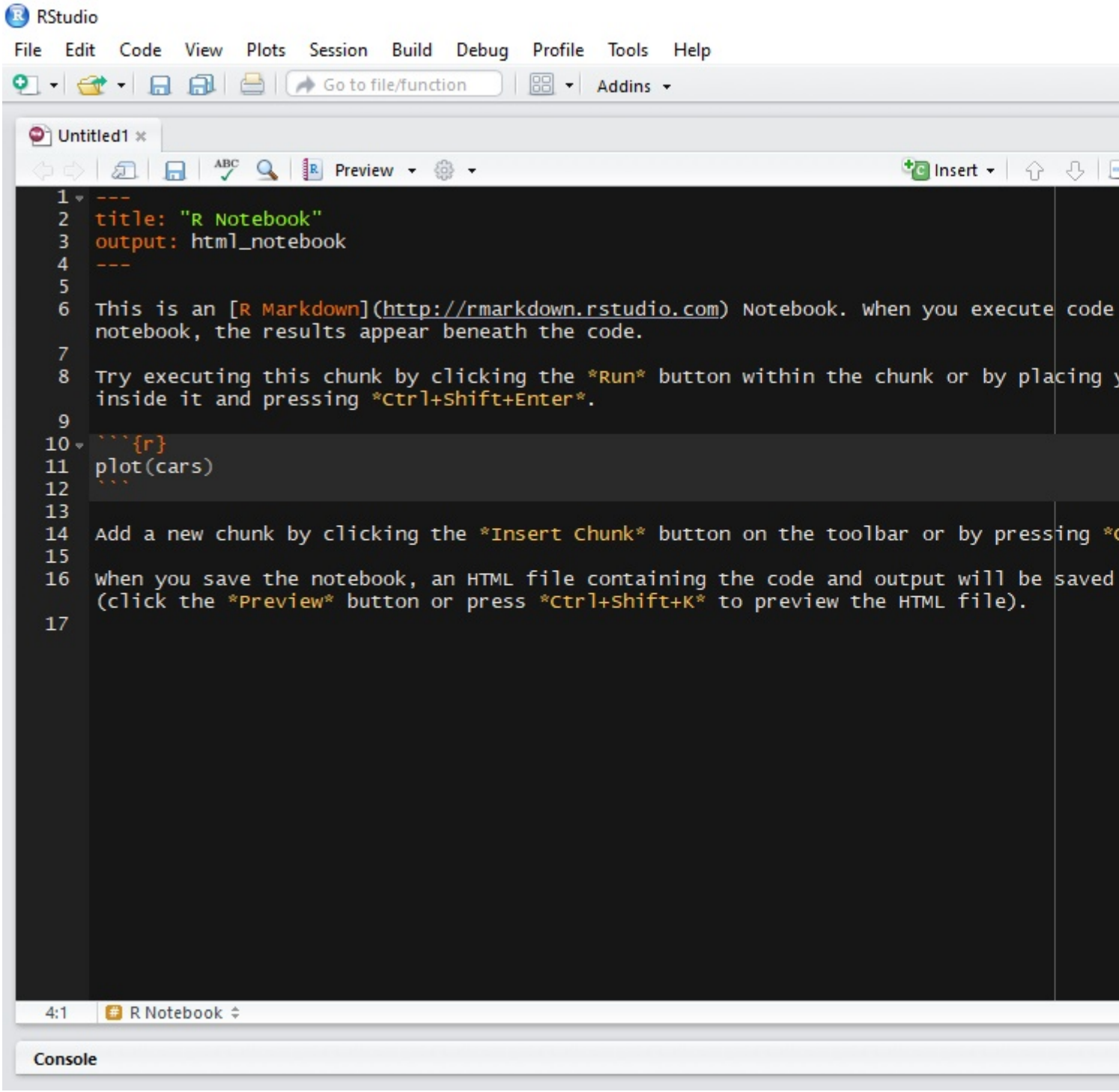
Splitting Code into Chunks

Since a chunk produces its output beneath the chunk, when having multiple lines of code in a

www.dbooks.org

single chunk that produces multiples outputs it is often helpful to split into multiple chunks such that each chunk produces one output.

To do this, select the code to you want to split into a new chunk and press **Ctrl + Alt + I (OS X: Cmd + Option + I)**



Execution Progress

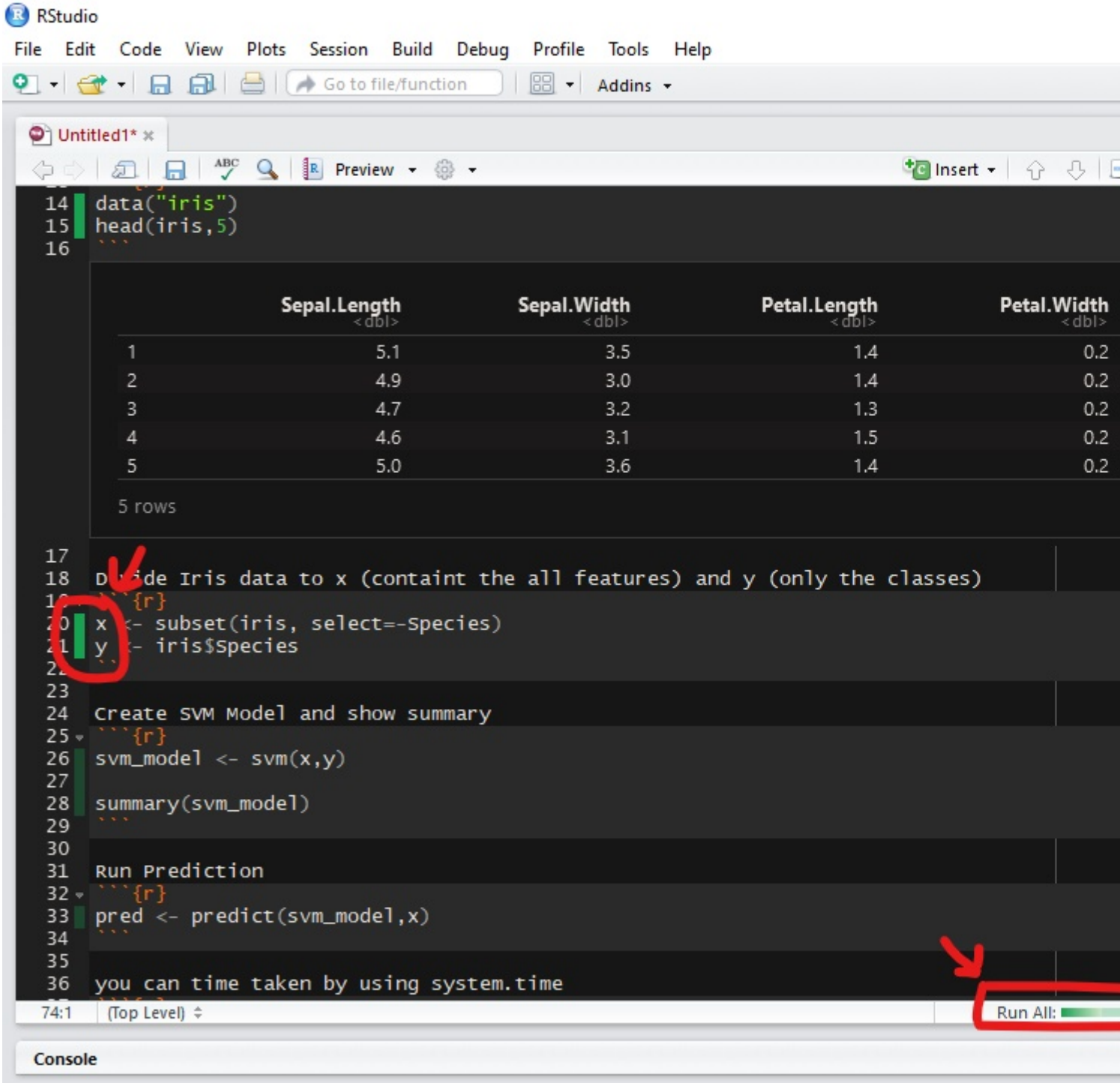
When you execute code in a notebook, an indicator will appear in the gutter to show you execution progress. Lines of code which have been sent to R are marked with dark green; lines which have not yet been sent to R are marked with light green.

Executing Multiple Chunks

Running or Re-Running individual chunks by pressing Run for all the chunks present in a document can be painful. We can use **Run All** from the Insert menu in the toolbar to Run all the chunks present in the notebook. Keyboard shortcut is **Ctrl + Alt + R (OS X: Cmd + Option + R)**

There's also a option **Restart R and Run All Chunks** command (available in the Run menu on the editor toolbar), which gives you a fresh R session prior to running all the chunks.

We also have options like **Run All Chunks Above** and **Run All Chunks Below** to run chunks Above or Below from a selected chunk.

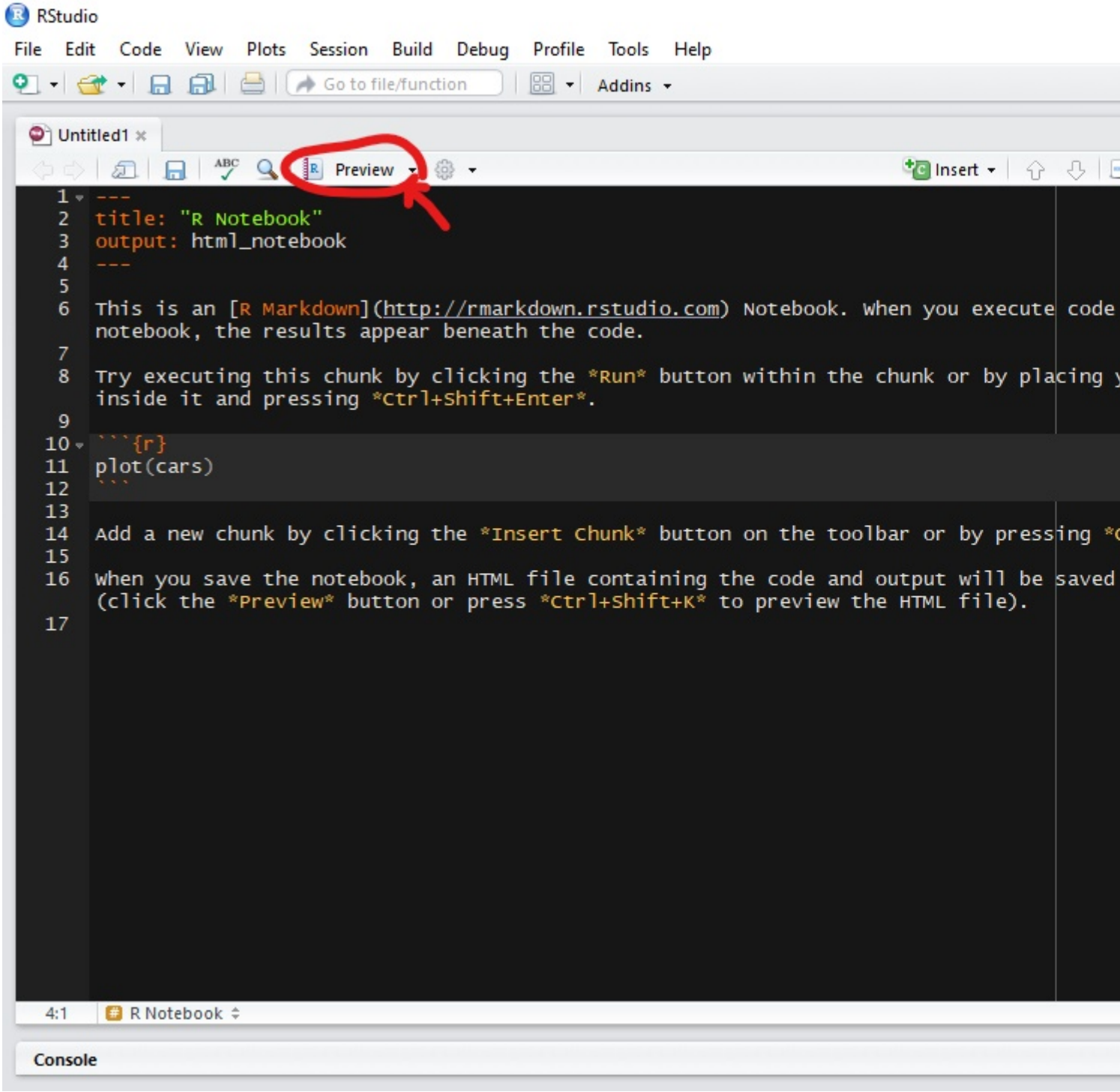


www.dbooks.org

Preview Output

Before rendering the final version of a notebook we can preview the output. Click on the **Preview** button on the toolbar and select the desired output format.

You can change the type of output by using the output options as "pdf_document" or "html_notebook"



Saving and Sharing

When a notebook `.Rmd` is saved, an `.nb.html` file is created alongside it. This file is a self-contained HTML file which contains both a rendered copy of the notebook with all current chunk outputs

(suitable for display on a website) and a copy of the notebook .Rmd itself.

More info can be found at RStudio docs

Read R Markdown Notebooks (from RStudio) online: https://riptutorial.com/r/topic/10728/r-markdown-notebooks--from-rstudio-

www.dbooks.org

Chapter 84: R memento by examples

Introduction

This topic is meant to be a memento about the R language without any text, with self-explanatory examples.

Each example is meant to be as succint as possible.

Examples

Data types

Vectors

```
a <- c(1, 2, 3)
b <- c(4, 5, 6)
mean_ab <- (a + b) / 2

d <- c(1, 0, 1)
only_1_3 <- a[d == 1]
```

Matrices

```
mat <- matrix(c(1,2,3,4), nrow = 2, ncol = 2)
dimnames(mat) <- list(c(), c("a", "b", "c"))
mat[,] == mat
```

Dataframes

```
df <- data.frame(qualifiers = c("Buy", "Sell", "Sell"),
                 symbols = c("AAPL", "MSFT", "GOOGL"),
                 values = c(326.0, 598.3, 201.5))
df$symbols == df[[2]]
df$symbols == df[["symbols"]]
df[[2, 1]] == "AAPL"
```

Lists

```
l <- list(a = 500, "aaa", 98.2)
length(l)       == 3
class(l[1])     == "list"
```

```
class(l[[1]])    == "numeric"
class(l$a)       == "numeric"
```

Environments

```
env <- new.env()
env[["foo"]] = "bar"
env2 <- env
env2[["foo"]] = "BAR"

env[["foo"]] == "BAR"
get("foo", envir = env) == "BAR"
rm("foo", envir = env)
env[["foo"]] == NULL
```

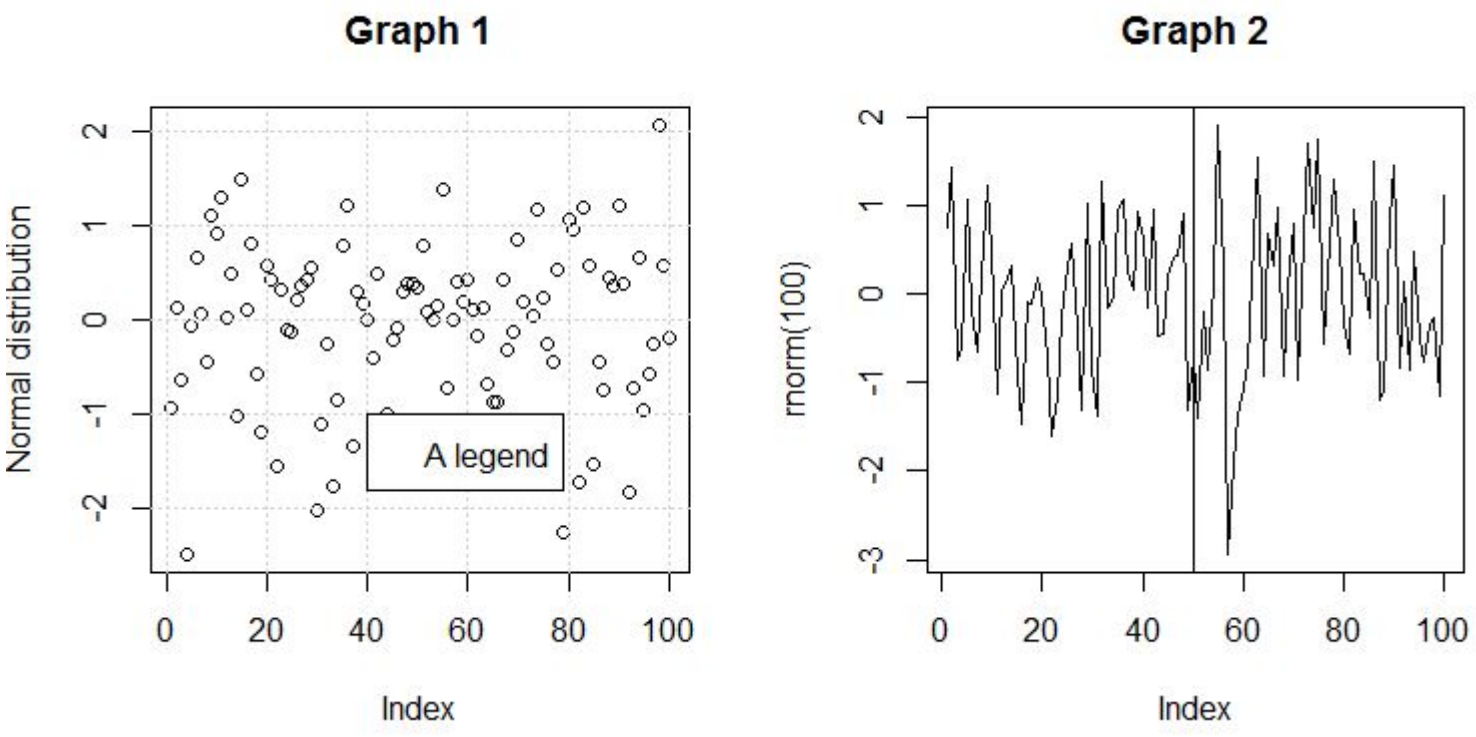
Plotting (using plot)

```
# Creates a 1 row - 2 columns format
par(mfrow=c(1,2))

plot(rnorm(100), main = "Graph 1", ylab = "Normal distribution")
grid()
legend(x = 40, y = -1, legend = "A legend")

plot(rnorm(100), main = "Graph 2", type = "l")
abline(v = 50)
```

Result:



Commonly used functions

www.dbooks.org

```
# Create 100 standard normals in a vector
x <- rnorm(100, mean = 0, sd = 1)

# Find the lenght of a vector
length(x)

# Compute the mean
mean(x)

# Compute the standard deviation
sd(x)

# Compute the median value
median(x)

# Compute the range (min, max)
range(x)

# Sum an iterable
sum(x)

# Cumulative sum (x[1], x[1]+x[2], ...)
cumsum(x)

# Display the first 3 elements
head(3, x)

# Display min, 1st quartile, median, mean, 3rd quartile, max
summary(x)

# Compute successive difference between elements
diff(x)

# Create a range from 1 to 10 step 1
1:10

# Create a range from 1 to 10 step 0.1
seq(1, 10, 0.1)

# Print a string
print("hello world")
```

Read R memento by examples online: https://riptutorial.com/r/topic/10827/r-memento-by-examples

Chapter 85: Random Forest Algorithm

Introduction

RandomForest is an ensemble method for classification or regression that reduces the chance of overfitting the data. Details of the method can be found in the Wikipedia article on Random Forests. The main implementation for R is in the randomForest package, but there are other implementations. See the CRAN view on Machine Learning.

Examples

Basic examples - Classification and Regression

```
######  Used for both Classification and Regression examples
library(randomForest)
library(car)            ## For the Soils data
data(Soils)

#####################################################
##    RF Classification Example
set.seed(656)            ## for reproducibility
S_RF_Class = randomForest(Gp ~ ., data=Soils[,c(4,6:14)])
Gp_RF = predict(S_RF_Class, Soils[,6:14])
length(which(Gp_RF != Soils$Gp))            ## No Errors

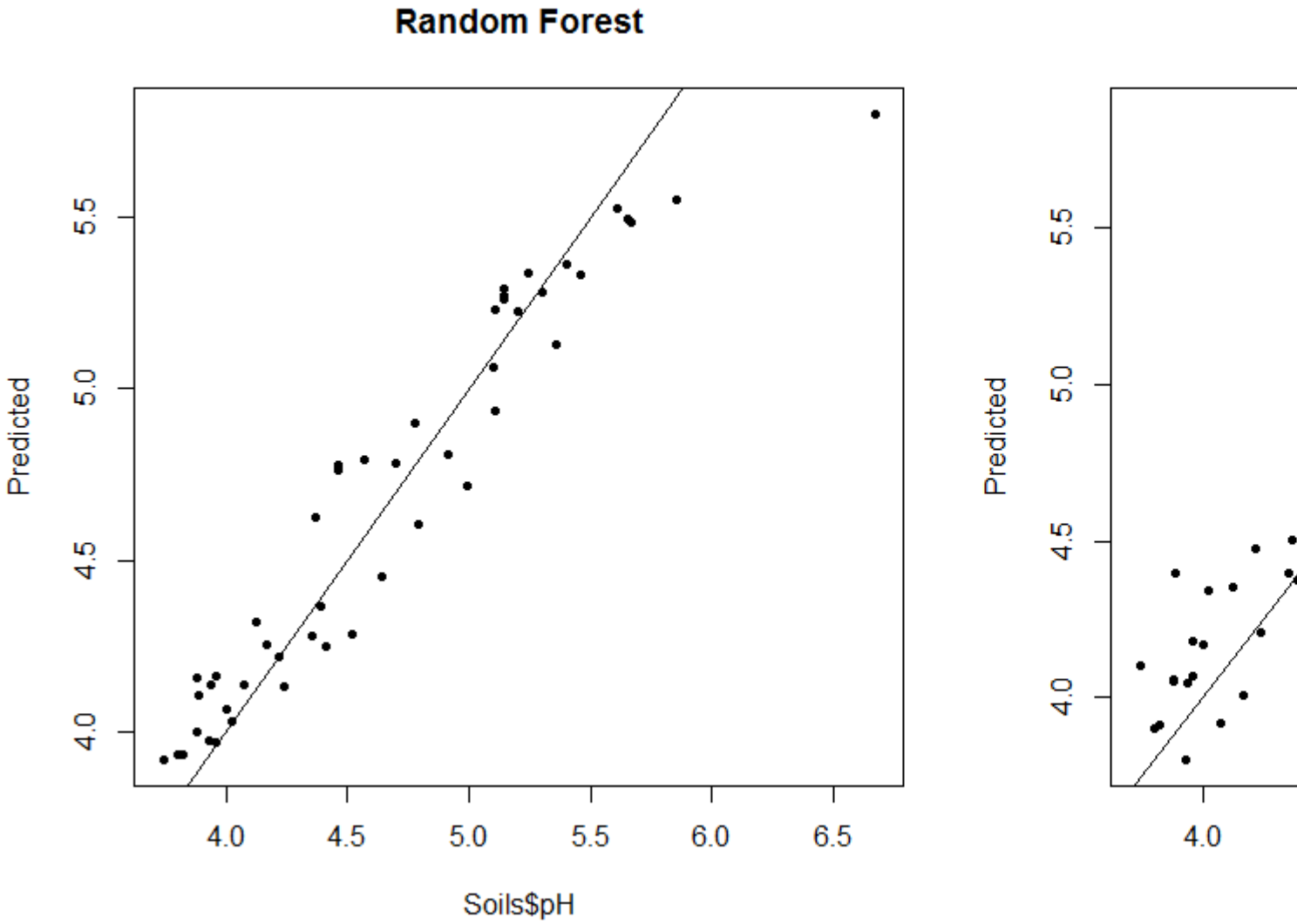
## Naive Bayes for comparison
library(e1071)
S_NB  = naiveBayes(Soils[,6:14], Soils[,4])
Gp_NB = predict(S_NB, Soils[,6:14], type="class")
length(which(Gp_NB != Soils$Gp))            ## 6 Errors
```

This example tested on the training data, but illustrates that RF can make very good models.

```
#####################################################
##    RF Regression Example
set.seed(656)            ## for reproducibility
S_RF_Reg = randomForest(pH ~ ., data=Soils[,6:14])
pH_RF = predict(S_RF_Reg, Soils[,6:14])

## Compare Predictions with Actual values for RF and Linear Model
S_LM = lm(pH ~ ., data=Soils[,6:14])
pH_LM = predict(S_LM, Soils[,6:14])
par(mfrow=c(1,2))
plot(Soils$pH, pH_RF, pch=20, ylab="Predicted", main="Random Forest")
abline(0,1)
plot(Soils$pH, pH_LM, pch=20, ylab="Predicted", main="Linear Model")
abline(0,1)
```


www.dbooks.org



Read Random Forest Algorithm online: https://riptutorial.com/r/topic/8088/random-forest-algorithm

Chapter 86: Random Numbers Generator

Examples

Random permutations

To generate random permutation of 5 numbers:

```
sample(5)
# [1] 4 5 3 1 2
```

To generate random permutation of any vector:

```
sample(10:15)
# [1] 11 15 12 10 14 13
```

One could also use the package `pracma`

```
randperm(a, k)
# Generates one random permutation of k of the elements a, if a is a vector,
# or of 1:a if a is a single integer.
# a: integer or numeric vector of some length n.
# k: integer, smaller as a or length(a).

# Examples
library(pracma)
randperm(1:10, 3)
[1] 3 7 9

randperm(10, 10)
[1]  4  5 10  8  2  7  6  9  3  1

randperm(seq(2, 10, by=2))
[1]  6  4 10  2  8
```

Random number generator's reproducibility

When expecting someone to reproduce an R code that has random elements in it, the `set.seed()` function becomes very handy. For example, these two lines will always produce different output (because that is the whole point of random number generators):

```
> sample(1:10,5)
[1]  6  9  2  7 10
> sample(1:10,5)
[1]  7  6  1  2 10
```

These two will also produce different outputs:

```
> rnorm(5)
```


www.dbooks.org

```
[1]  0.4874291  0.7383247  0.5757814 -0.3053884  1.5117812
> rnorm(5)
[1]  0.38984324 -0.62124058 -2.21469989  1.12493092 -0.04493361
```

However, if we set the seed to something identical in both cases (most people use 1 for simplicity), we get two identical samples:

```
> set.seed(1)
> sample(letters,2)
[1] "g" "j"
> set.seed(1)
> sample(letters,2)
[1] "g" "j"
```

and same with, say, `rexp()` draws:

```
> set.seed(1)
> rexp(5)
[1] 0.7551818 1.1816428 0.1457067 0.1397953 0.4360686
> set.seed(1)
> rexp(5)
[1] 0.7551818 1.1816428 0.1457067 0.1397953 0.4360686
```

Generating random numbers using various density functions

Below are examples of generating 5 random numbers using various probability distributions.

Uniform distribution between 0 and 10

```
runif(5, min=0, max=10)
[1] 2.1724399 8.9209930 6.1969249 9.3303321 2.4054102
```

Normal distribution with 0 mean and standard deviation of 1

```
rnorm(5, mean=0, sd=1)
[1] -0.97414402 -0.85722281 -0.08555494 -0.37444299  1.20032409
```

Binomial distribution with 10 trials and success probability of 0.5

```
rbinom(5, size=10, prob=0.5)
[1] 4 3 5 2 3
```

Geometric distribution with 0.2 success probability

```
rgeom(5, prob=0.2)
```

```
[1] 14  8 11  1  3
```

Hypergeometric distribution with 3 white balls, 10 black balls and 5 draws

```
rhyper(5, m=3, n=10, k=5)
[1] 2 0 1 1 1
```

Negative Binomial distribution with 10 trials and success probability of 0.8

```
rnbinom(5, size=10, prob=0.8)
[1] 3 1 3 4 2
```

Poisson distribution with mean and variance (lambda) of 2

```
rpois(5, lambda=2)
[1] 2 1 2 3 4
```

Exponential distribution with the rate of 1.5

```
rexp(5, rate=1.5)
[1] 1.8993303 0.4799358 0.5578280 1.5630711 0.6228000
```

Logistic distribution with 0 location and scale of 1

```
rlogis(5, location=0, scale=1)
[1]  0.9498992 -1.0287433 -0.4192311  0.7028510 -1.2095458
```

Chi-squared distribution with 15 degrees of freedom

```
rchisq(5, df=15)
[1] 14.89209 19.36947 10.27745 19.48376 23.32898
```

Beta distribution with shape parameters a=1 and b=0.5

```
rbeta(5, shape1=1, shape2=0.5)
[1] 0.1670306 0.5321586 0.9869520 0.9548993 0.9999737
```

Gamma distribution with shape parameter of 3 and scale=0.5

www.dbooks.org

```
rgamma(5, shape=3, scale=0.5)
[1] 2.2445984 0.7934152 3.2366673 2.2897537 0.8573059
```

Cauchy distribution with 0 location and scale of 1

```
rcauchy(5, location=0, scale=1)
[1] -0.01285116 -0.38918446  8.71016696 10.60293284 -0.68017185
```

Log-normal distribution with 0 mean and standard deviation of 1 (on log scale)

```
rlnorm(5, meanlog=0, sdlog=1)
[1] 0.8725009 2.9433779 0.3329107 2.5976206 2.8171894
```

Weibull distribution with shape parameter of 0.5 and scale of 1

```
rweibull(5, shape=0.5, scale=1)
[1] 0.337599112 1.307774557 7.233985075 5.840429942 0.005751181
```

Wilcoxon distribution with 10 observations in the first sample and 20 in second.

```
rwilcox(5, 10, 20)
[1] 111  88  93 100 124
```

Multinomial distribution with 5 object and 3 boxes using the specified probabilities

```
rmultinom(5, size=5, prob=c(0.1,0.1,0.8))
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    1    1    0
[2,]    2    0    1    1    0
[3,]    3    5    3    3    5
```

Read Random Numbers Generator online: https://riptutorial.com/r/topic/1578/random-numbers-generator

Chapter 87: Randomization

Introduction

The R language is commonly used for statistical analysis. As such, it contains a robust set of options for randomization. For specific information on sampling from probability distributions, see the documentation for distribution functions.

Remarks

Users who are coming from other programming languages may be confused by the lack of a `rand` function equivalent to what they may have experienced before. Basic random number generation is done using the `r*` family of functions for each distribution (see the link above). Random numbers drawn uniformly from a range can be generated using `runif`, for "random uniform". Since this also looks suspiciously like "run if", it is often hard to figure out for new R users.

Examples

Random draws and permutations

The `sample` command can be used to simulate classic probability problems like drawing from an urn with and without replacement, or creating random permutations.

Note that throughout this example, `set.seed` is used to ensure that the example code is reproducible. However, `sample` will work without explicitly calling `set.seed`.

Random permutation

In the simplest form, `sample` creates a random permutation of a vector of integers. This can be accomplished with:

```
set.seed(1251)
sample(x = 10)

[1]  7  1  4  8  6  3 10  5  2  9
```

When given no other arguments, `sample` returns a random permutation of the vector from 1 to `x`. This can be useful when trying to randomize the order of the rows in a data frame. This is a common task when creating randomization tables for trials, or when selecting a random subset of rows for analysis.

```
library(datasets)
set.seed(1171)
iris_rand <- iris[sample(x = 1:nrow(iris)),]
```


www.dbooks.org

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa

> head(iris_rand)
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
145          6.7         3.3          5.7         2.5  virginica
5            5.0         3.6          1.4         0.2     setosa
85           5.4         3.0          4.5         1.5 versicolor
137          6.3         3.4          5.6         2.4  virginica
128          6.1         3.0          4.9         1.8  virginica
105          6.5         3.0          5.8         2.2  virginica
```

Draws without Replacement

Using `sample`, we can also simulate drawing from a set with and without replacement. To sample without replacement (the default), you must provide sample with a set to be drawn from and the number of draws. The set to be drawn from is given as a vector.

```
set.seed(7043)
sample(x = LETTERS,size = 7)

[1] "S" "P" "J" "F" "Z" "G" "R"
```

Note that if the argument to `size` is the same as the length of the argument to `x`, you are creating a random permutation. Also note that you cannot specify a size greater than the length of `x` when doing sampling without replacement.

```
set.seed(7305)
sample(x = letters,size = 26)

[1] "x" "z" "y" "i" "k" "f" "d" "s" "g" "v" "j" "o" "e" "c" "m" "n" "h" "u" "a" "b" "l" "r"
"w" "t" "q" "p"

sample(x = letters,size = 30)
Error in sample.int(length(x), size, replace, prob) :
  cannot take a sample larger than the population when 'replace = FALSE'
```

This brings us to drawing with replacement.

Draws with Replacement

To make random draws from a set with replacement, you use the `replace` argument to `sample`. By default, `replace` is `FALSE`. Setting it to `TRUE` means that each element of the set being drawn from may appear more than once in the final result.

```
set.seed(5062)
sample(x = c("A","B","C","D"),size = 8,replace = TRUE)

[1] "D" "C" "D" "B" "A" "A" "A" "A"
```

Changing Draw Probabilities

By default, when you use `sample`, it assumes that the probability of picking each element is the same. Consider it as a basic "urn" problem. The code below is equivalent to drawing a colored marble out of an urn 20 times, writing down the color, and then putting the marble back in the urn. The urn contains one red, one blue, and one green marble, meaning that the probability of drawing each color is 1/3.

```
set.seed(6472)
sample(x = c("Red","Blue","Green"),
       size = 20,
       replace = TRUE)
```

Suppose that, instead, we wanted to perform the same task, but our urn contains 2 red marbles, 1 blue marble, and 1 green marble. One option would be to change the argument we send to `x` to add an additional `Red`. However, a better choice is to use the `prob` argument to `sample`.

The `prob` argument accepts a vector with the probability of drawing each element. In our example above, the probability of drawing a red marble would be 1/2, while the probability of drawing a blue or a green marble would be 1/4.

```
set.seed(28432)
sample(x = c("Red","Blue","Green"),
       size = 20,
       replace = TRUE,
       prob = c(0.50,0.25,0.25))
```

Counter-intuitively, the argument given to `prob` does not need to sum to 1. R will always transform the given arguments into probabilities that total to 1. For instance, consider our above example of 2 Red, 1 Blue, and 1 Green. You can achieve the same results as our previous code using those numbers:

```
set.seed(28432)
frac_prob_example <- sample(x = c("Red","Blue","Green"),
                            size = 200,
                            replace = TRUE,
                            prob = c(0.50,0.25,0.25))

set.seed(28432)
numeric_prob_example <- sample(x = c("Red","Blue","Green"),
                               size = 200,
                               replace = TRUE,
                               prob = c(2,1,1))

> identical(frac_prob_example,numeric_prob_example)
[1] TRUE
```


www.dbooks.org

The major restriction is that you cannot set all the probabilities to be zero, and none of them can be less than zero.

You can also utilize `prob` when `replace` is set to `FALSE`. In that situation, after each element is drawn, the proportions of the `prob` values for the remaining elements give the probability for the next draw. In this situation, you must have enough non-zero probabilities to reach the `size` of the sample you are drawing. For example:

```
set.seed(21741)
sample(x = c("Red","Blue","Green"),
       size = 2,
       replace = FALSE,
       prob = c(0.8,0.19,0.01))
```

In this example, Red is drawn in the first draw (as the first element). There was an 80% chance of Red being drawn, a 19% chance of Blue being drawn, and a 1% chance of Green being drawn.

For the next draw, Red is no longer in the urn. The total of the probabilities among the remaining items is 20% (19% for Blue and 1% for Green). For that draw, there is a 95% chance the item will be Blue (19/20) and a 5% chance it will be Green (1/20).

Setting the seed

The `set.seed` function is used to set the random seed for all randomization functions. If you are using R to create a randomization that you want to be able to reproduce, you should use `set.seed` first.

```
set.seed(1643)
samp1 <- sample(x = 1:5,size = 200,replace = TRUE)

set.seed(1643)
samp2 <- sample(x = 1:5,size = 200,replace = TRUE)

> identical(x = samp1,y = samp2)
[1] TRUE
```

Note that parallel processing requires special treatment of the random seed, described more elsewhere.

Read Randomization online: https://riptutorial.com/r/topic/9574/randomization

Chapter 88: Raster and Image Analysis

Introduction

See also I/O for Raster Images

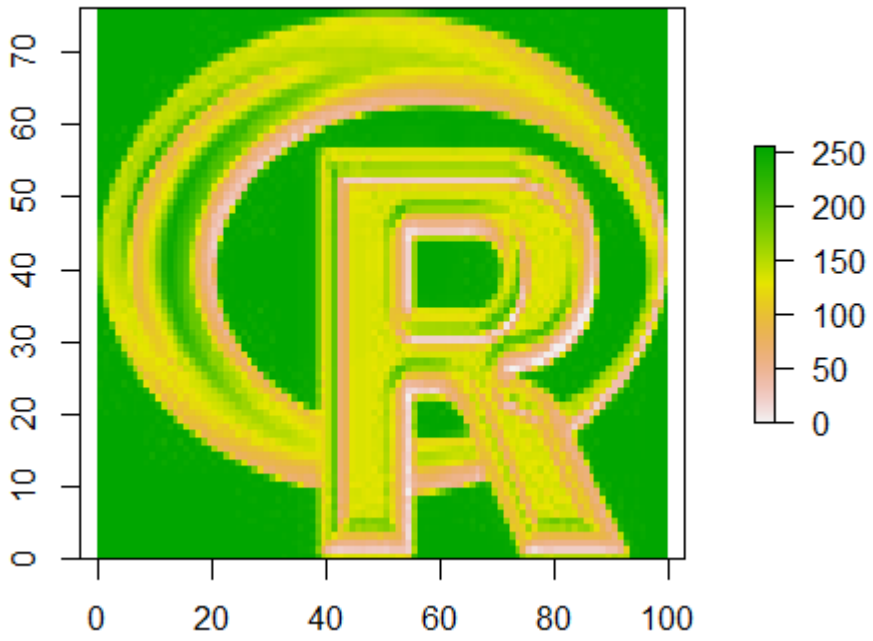
Examples

Calculating GLCM Texture

Gray Level Co-Occurrence Matrix (Haralick et al. 1973) texture is a powerful image feature for image analysis. The `glcm` package provides a easy-to-use function to calculate such texutral features for `RasterLayer` objects in R.

```
library(glcm)
library(raster)

r <- raster("C:/Program Files/R/R-3.2.3/doc/html/logo.jpg")
plot(r)
```

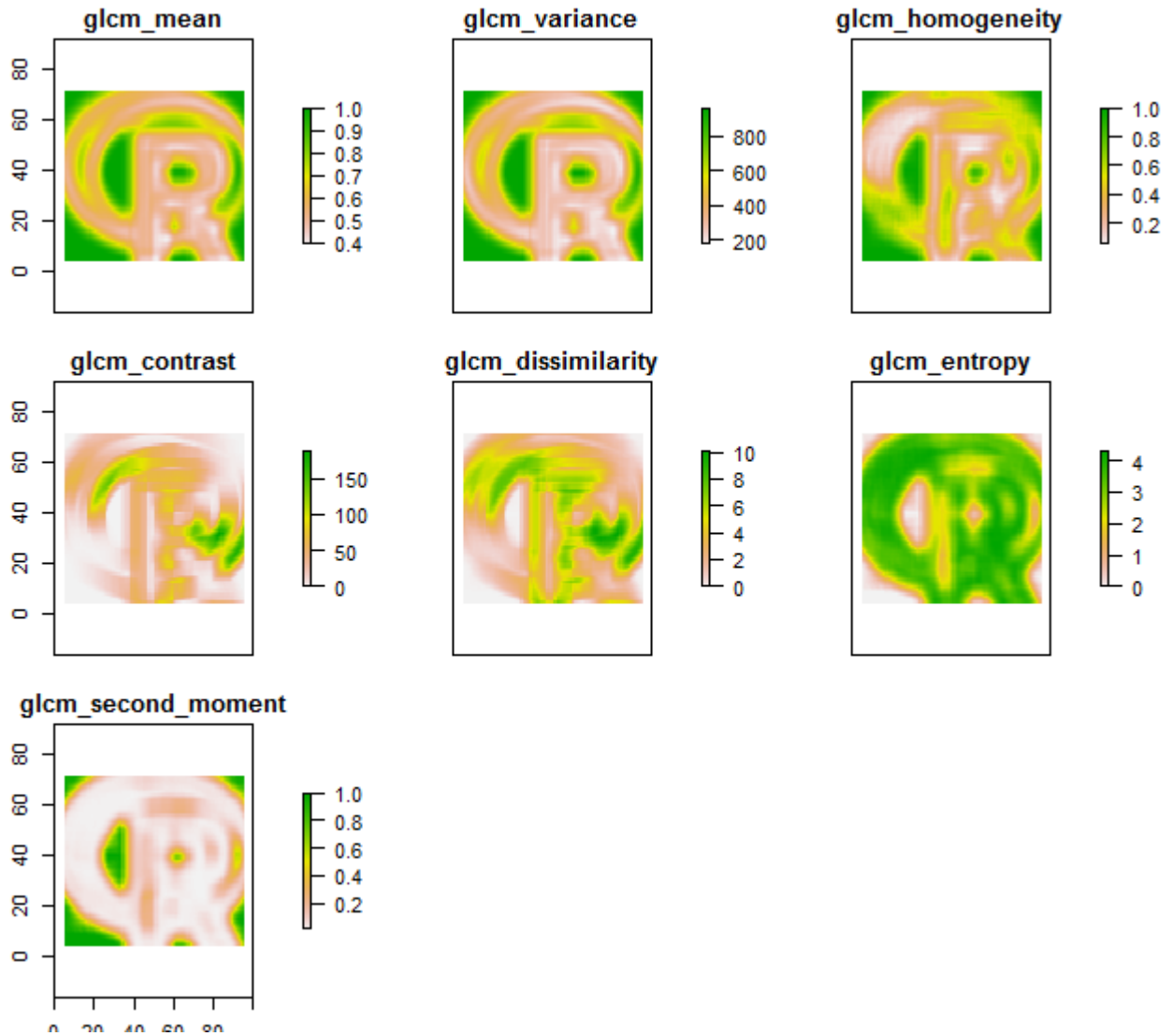


Calculating GLCM textures in one direction

```
rglcm <- glcm(r,
              window = c(9,9),
              shift = c(1,1),
              statistics = c("mean", "variance", "homogeneity", "contrast",
                             "dissimilarity", "entropy", "second_moment")
              )
```


www.dbooks.org

```
plot(rglcm)
```

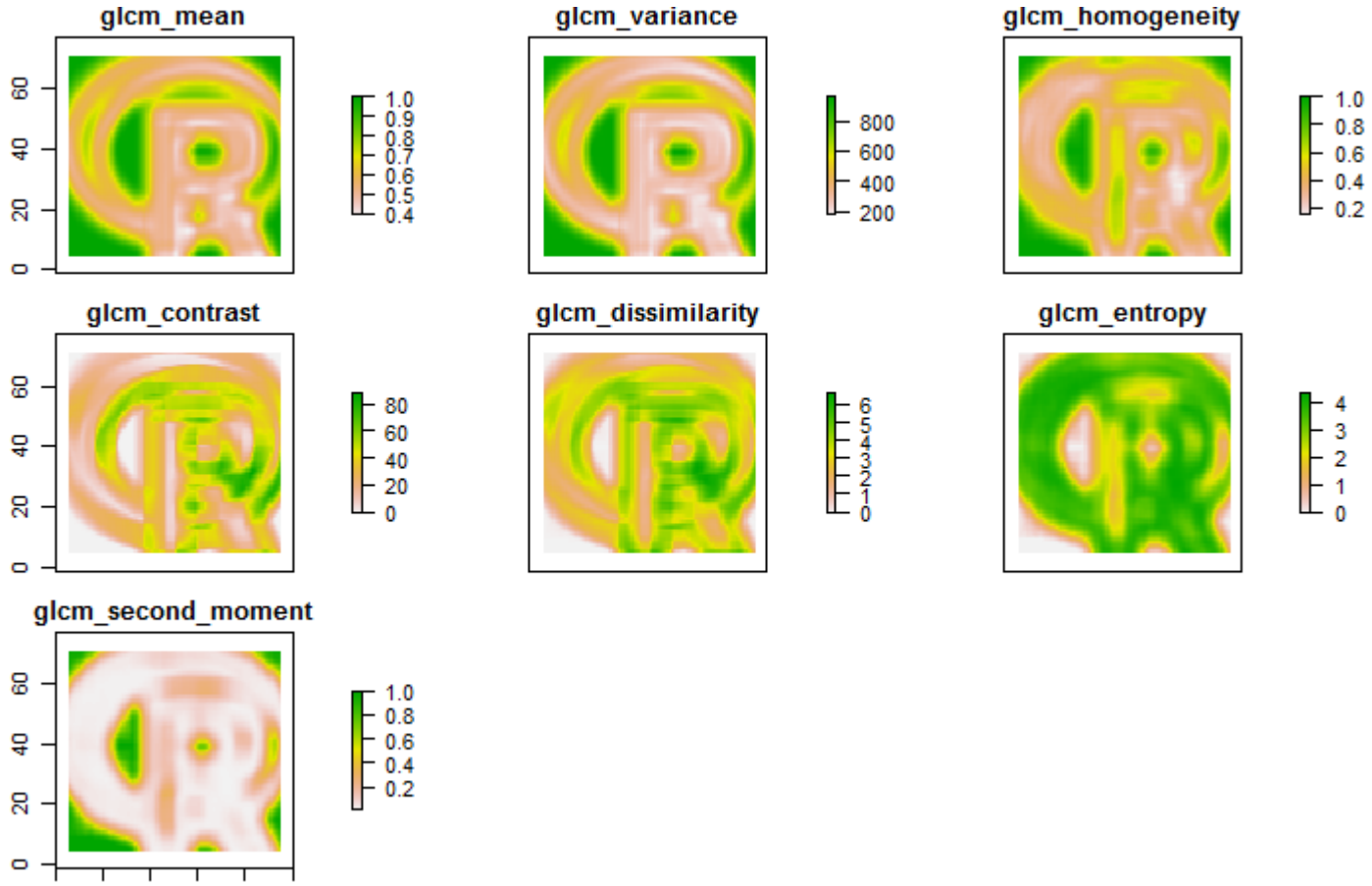


Calculation rotation-invariant texture features

The textural features can also be calculated in all 4 directions (0°, 45°, 90° and 135°) and then combined to one rotation-invariant texture. The key for this is the `shift` parameter:

```
rglcm1 <- glcm(r,
               window = c(9,9),
               shift=list(c(0,1), c(1,1), c(1,0), c(1,-1)),
               statistics = c("mean", "variance", "homogeneity", "contrast",
                              "dissimilarity", "entropy", "second_moment")
               )

plot(rglcm1)
```



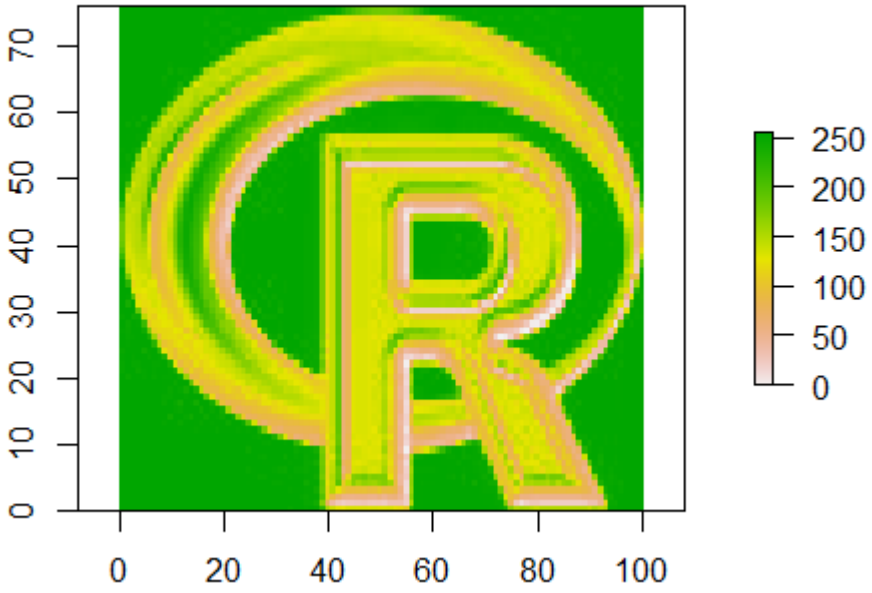
Mathematical Morphologies

The package `mmand` provides functions for the calculation of Mathematical Morphologies for n-dimensional arrays. With a little workaround, these can also be calculated for raster images.

```
library(raster)
library(mmand)

r <- raster("C:/Program Files/R/R-3.2.3/doc/html/logo.jpg")
plot(r)
```


www.dbooks.org



At first, a kernel (moving window) has to be set with a size (e.g. 9x9) and a shape type (e.g. `disc`, `box` or `diamond`)

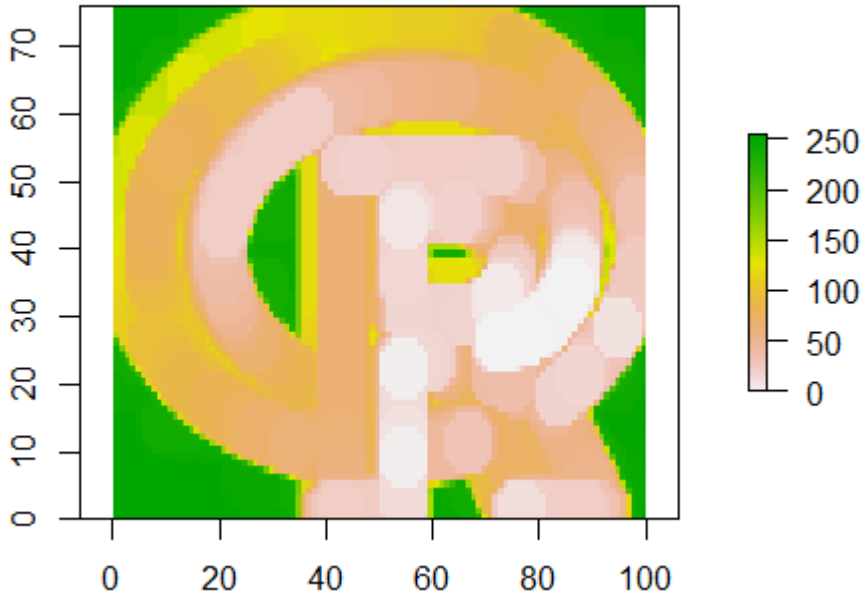
```
sk <- shapeKernel(c(9,9), type="disc")
```

Afterwards, the raster layer has to be converted into an array wich is used as input for the `erode()` function.

```
rArr <- as.array(r, transpose = TRUE)
rErode <- erode(rArr, sk)
rErode <- setValues(r, as.vector(aperm(rErode)))
```

Besides `erode()`, also the morphological functions `dilate()`, `opening()` and `closing()` can be applied like this.

```
plot(rErode)
```



Read Raster and Image Analysis online: https://riptutorial.com/r/topic/3726/raster-and-image-analysis

www.dbooks.org

Chapter 89: Rcpp

Examples

Inline Code Compile

Rcpp features two functions that enable code compilation inline and exportation directly into R: `cppFunction()` and `evalCpp()`. A third function called `sourceCpp()` exists to read in C++ code in a separate file though can be used akin to `cppFunction()`.

Below is an example of compiling a C++ function within *R*. Note the use of `""` to surround the source.

```
# Note - This is R code.
# cppFunction in Rcpp allows for rapid testing.
require(Rcpp)

# Creates a function that multiples each element in a vector
# Returns the modified vector.
cppFunction("
NumericVector exfun(NumericVector x, int i){
x = x*i;
return x;
}")

# Calling function in R
exfun(1:5, 3)
```

To quickly understand a C++ expression use:

```
# Use evalCpp to evaluate C++ expressions
evalCpp("std::numeric_limits<double>::max()")
## [1] 1.797693e+308
```

Rcpp Attributes

Rcpp Attributes makes the process of working with R and C++ straightforward. The form of attributes take:

```
// [[Rcpp::attribute]]
```

The use of attributes is typically associated with:

```
// [[Rcpp::export]]
```

that is placed directly above a declared function header when reading in a C++ file via `sourceCpp()`.

Below is an example of an external C++ file that uses attributes.

```
// Add code below into C++ file Rcpp_example.cpp

#include <Rcpp.h>
using namespace Rcpp;

// Place the export tag right above function declaration.
// [[Rcpp::export]]
double muRcpp(NumericVector x){

    int n = x.size(); // Size of vector
    double sum = 0; // Sum value

    // For loop, note cpp index shift to 0
    for(int i = 0; i < n; i++){
        // Shorthand for sum = sum + x[i]
        sum += x[i];
    }

    return sum/n; // Obtain and return the Mean
}

// Place dependent functions above call or
// declare the function definition with:
double muRcpp(NumericVector x);

// [[Rcpp::export]]
double varRcpp(NumericVector x, bool bias = true){

    // Calculate the mean using C++ function
    double mean = muRcpp(x);
    double sum = 0;

    int n = x.size();

    for(int i = 0; i < n; i++){
        sum += pow(x[i] - mean, 2.0); // Square
    }

    return sum/(n-bias); // Return variance
}
```

To use this external *C++* file within *R*, we do the following:

```
require(Rcpp)

# Compile File
sourceCpp("path/to/file/Rcpp_example.cpp")

# Make some sample data
x = 1:5

all.equal(muRcpp(x), mean(x))
## TRUE

all.equal(varRcpp(x), var(x))
## TRUE
```

Extending Rcpp with Plugins

www.dbooks.org

Within C++, one can set different compilation flags using:

```
 // [[Rcpp::plugins(name)]]
```

List of the built-in plugins:

```
// built-in C++11 plugin
// [[Rcpp::plugins(cpp11)]]

// built-in C++11 plugin for older g++ compiler
// [[Rcpp::plugins(cpp0x)]]

// built-in C++14 plugin for C++14 standard
// [[Rcpp::plugins(cpp14)]]

// built-in C++1y plugin for C++14 and C++17 standard under development
// [[Rcpp::plugins(cpp1y)]]

// built-in OpenMP++11 plugin
// [[Rcpp::plugins(openmp)]]
```

Specifying Additional Build Dependencies

To use additional packages within the Rcpp ecosystem, the correct header file may not be `Rcpp.h` but `Rcpp<PACKAGE>.h` (as *e.g.* for RcppArmadillo). It typically needs to be imported and then the dependency is stated within

```
// [[Rcpp::depends(Rcpp<PACKAGE>)]]
```

Examples:

```
// Use the RcppArmadillo package
// Requires different header file from Rcpp.h
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]

// Use the RcppEigen package
// Requires different header file from Rcpp.h
#include <RcppEigen.h>
// [[Rcpp::depends(RcppEigen)]]
```

Read Rcpp online: https://riptutorial.com/r/topic/1404/rcpp

Chapter 90: Reading and writing strings

Remarks

Related Docs:

- Get user input

Examples

Printing and displaying strings

R has several built-in functions that can be used to print or display information, but `print` and `cat` are the most basic. As R is an interpreted language, you can try these out directly in the R console:

```
print("Hello World")
#[1] "Hello World"
cat("Hello World\n")
#Hello World
```

Note the difference in both input and output for the two functions. (Note: there are no quote-characters in the value of `x` created with `x <- "Hello World"`. They are added by `print` at the output stage.)

`cat` takes one or more character vectors as arguments and prints them to the console. If the character vector has a length greater than 1, arguments are separated by a space (by default):

```
cat(c("hello", "world", "\n"))
#hello world
```

Without the new-line character (`\n`) the output would be:

```
cat("Hello World")
#Hello World>
```

The prompt for the next command appears immediately after the output. (Some consoles such as RStudio's may automatically append a newline to strings that do not end with a newline.)

`print` is an example of a "generic" function, which means the class of the first argument passed is detected and a class-specific *method* is used to output. For a character vector like `"Hello World"`, the result is similar to the output of `cat`. However, the character string is quoted and a number `[1]` is output to indicate the first element of a character vector (In this case, the first and only element):

```
print("Hello World")
#[1] "Hello World"
```


www.dbooks.org

This default print method is also what we see when we simply ask R to print a variable. Note how the output of typing `s` is the same as calling `print(s)` or `print("Hello World")`:

```
s <- "Hello World"
s
#[1] "Hello World"
```

Or even without assigning it to anything:

```
"Hello World"
#[1] "Hello World"
```

If we add another character string as a second element of the vector (using the `c()` function to **c** oncatenate the elements together), then the behavior of `print()` looks quite a bit different from that of `cat`:

```
print(c("Hello World", "Here I am."))
#[1] "Hello World" "Here I am."
```

Observe that the `c()` function does *not* do string-concatenation. (One needs to use `paste` for that purpose.) R shows that the character vector has two elements by quoting them separately. If we have a vector long enough to span multiple lines, R will print the index of the element starting each line, just as it prints `[1]` at the start of the first line.

```
c("Hello World", "Here I am!", "This next string is really long.")
#[1] "Hello World"                    "Here I am!"
#[3] "This next string is really long."
```

The particular behavior of `print` depends on the *class* of the object passed to the function.

If we call `print` an object with a different class, such as "numeric" or "logical", the quotes are omitted from the output to indicate we are dealing with an object that is not character class:

```
print(1)
#[1] 1
print(TRUE)
#[1] TRUE
```

Factor objects get printed in the same fashion as character variables which often creates ambiguity when console output is used to display objects in SO question bodies. It is rare to use `cat` or `print` except in an interactive context. Explicitly calling `print()` is particularly rare (unless you wanted to suppress the appearance of the quotes or view an object that is returned as `invisible` by a function), as entering `foo` at the console is a shortcut for `print(foo)`. The interactive console of R is known as a REPL, a "read-eval-print-loop". The `cat` function is best saved for special purposes (like writing output to an open file connection). Sometimes it is used inside functions (where calls to `print()` are suppressed), however **using `cat()` inside a function to generate output to the console is bad practice**. The preferred method is to `message()` or `warning()` for intermediate messages; they behave similarly to `cat` but can be optionally suppressed by the end user. The final result should simply returned so that the user can assign it

to store it if necessary.

```
message("hello world")
#hello world
suppressMessages(message("hello world"))
```

Reading from or writing to a file connection

Not always we have liberty to read from or write to a local system path. For example if R code streaming map-reduce must need to read and write to file connection. There can be other scenarios as well where one is going beyond local system and with advent of cloud and big data, this is becoming increasingly common. One of the way to do this is in logical sequence.

Establish a file connection to read with `file()` command ("r" is for read mode):

```
conn <- file("/path/example.data", "r") #when file is in local system
conn1 <- file("stdin", "r") #when just standard input/output for files are available
```

As this will establish just file connection, one can read the data from these file connections as follows:

```
line <- readLines(conn, n=1, warn=FALSE)
```

Here we are reading the data from file connection `conn` line by line as `n=1`. one can change value of `n` (say 10, 20 etc.) for reading data blocks for faster reading (10 or 20 lines block read in one go). To read complete file in one go set `n=-1`.

After data processing or say model execution; one can write the results back to file connection using many different commands like `writeLines(),cat()` etc. which are capable of writing to a file connection. However all of these commands will leverage file connection established for writing. This could be done using `file()` command as:

```
conn2 <- file("/path/result.data", "w") #when file is in local system
conn3 <- file("stdout", "w") #when just standard input/output for files are available
```

Then write the data as follows:

```
writeLines("text",conn2, sep = "\n")
```

Capture output of operating system command

Functions which return a character vector

Base `R` has two functions for invoking a system command. Both require an additional parameter to capture the output of the system command.

www.dbooks.org

```
system("top -a -b -n 1", intern = TRUE)
system2("top", "-a -b -n 1", stdout = TRUE)
```

Both return a character vector.

```
  [1] "top - 08:52:03 up 70 days, 15:09,  0 users,  load average: 0.00, 0.00, 0.00"
  [2] "Tasks: 125 total,   1 running, 124 sleeping,   0 stopped,   0 zombie"
  [3] "Cpu(s):  0.9%us,  0.3%sy,  0.0%ni, 98.7%id,  0.1%wa,  0.0%hi,  0.0%si,  0.0%st"
  [4] "Mem:  12194312k total,  3613292k used,  8581020k free,   216940k buffers"
  [5] "Swap: 12582908k total,  2334156k used, 10248752k free,  1682340k cached"
  [6] ""
  [7] "  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND           "
  [8] "11300 root      20   0 1278m 375m 3696 S  0.0  3.2 124:40.92 trala             "
  [9] " 6093 user1     20   0 1817m 269m 1888 S  0.0  2.3  12:17.96 R                 "
 [10] " 4949 user2     20   0 1917m 214m 1888 S  0.0  1.8  11:16.73 R                 "
```

> For illustration, the UNIX command `top -a -b -n 1` is used. This is OS specific and may need to be amended to run the examples on your computer.

Package `devtools` has a function to run a system command and capture the output without an additional parameter. It also returns a character vector.

```
devtools::system_output("top", "-a -b -n 1")
```

Functions which return a data frame

The `fread` function in package `data.table` allows to execute a shell command and to read the output like `read.table`. It returns a `data.table` or a `data.frame`.

```
fread("top -a -b -n 1", check.names = TRUE)
       PID     USER PR NI  VIRT  RES  SHR S X.CPU X.MEM     TIME.         COMMAND
  1: 11300     root 20  0 1278m 375m 3696 S     0   3.2 124:40.92           trala
  2:  6093    user1 20  0 1817m 269m 1888 S     0   2.3  12:18.56               R
  3:  4949    user2 20  0 1917m 214m 1888 S     0   1.8  11:17.33               R
  4:  7922    user3 20  0 3094m 131m 1892 S     0   1.1  21:04.95               R
```

Note, that `fread` automatically has skipped the top 6 header lines.

> Here the parameter `check.names = TRUE` was added to convert `%CPU`, `%MEN`, and `TIME+` to syntactically valid column names.

Read Reading and writing strings online: https://riptutorial.com/r/topic/5541/reading-and-writing-strings

Chapter 91: Reading and writing tabular data in plain-text files (CSV, TSV, etc.)

Syntax

- read.csv(file, header = TRUE, sep = ",", quote = """, dec = ".", fill = TRUE, comment.char = "", ...)
- read.csv2(file, header = TRUE, sep = ";", quote = """, dec = ",", fill = TRUE, comment.char = "", ...)
- readr::read_csv(file, col_names = TRUE, col_types = NULL, locale = default_locale(), na = c("", "NA"), comment = "", trim_ws = TRUE, skip = 0, n_max = -1, progress = interactive())
- data.table::fread(input, sep="auto", sep2="auto", nrows=-1L, header="auto", na.strings="NA", stringsAsFactors=FALSE, verbose=getOption("datatable.verbose"), autostart=1L, skip=0L, select=NULL, drop=NULL, colClasses=NULL, integer64=getOption("datatable.integer64"), # default: "integer64" dec=if (sep!=".") "." else ",", col.names, check.names=FALSE, encoding="unknown", strip.white=TRUE, showProgress=getOption("datatable.showProgress"), # default: TRUE data.table=getOption("datatable.fread.datatable") # default: TRUE)

Parameters

Parameter	Details
file	name of the CSV file to read
header	logical: does the .csv file contain a header row with column names?
sep	character: symbol that separates the cells on each row
quote	character: symbol used to quote character strings
dec	character: symbol used as decimal separator
fill	logical: when TRUE, rows with unequal length are filled with blank fields.
comment.char	character: character used as comment in the csv file. Lines preceded by this character are ignored.
...	extra arguments to be passed to `read.table`

Remarks

www.dbooks.org

Note that exporting to a plain text format sacrifices much of the information encoded in the data like variable classes for the sake of wide portability. For cases that do not require such portability, a format like .RData or Feather may be more useful.

Input/output for other types of files is covered in several other topics, all linked from Input and output.

Examples

Importing .csv files

Importing using base R

Comma separated value files (CSVs) can be imported using `read.csv`, which wraps `read.table`, but uses `sep = ","` to set the delimiter to a comma.

```
# get the file path of a CSV included in R's utils package
csv_path <- system.file("misc", "exDIF.csv", package = "utils")

# path will vary based on installation location
csv_path
## [1] "/Library/Frameworks/R.framework/Resources/library/utils/misc/exDIF.csv"

df <- read.csv(csv_path)

df
##    Var1 Var2
## 1  2.70    A
## 2  3.14    B
## 3 10.00    A
## 4 -7.00    A
```

A user friendly option, `file.choose`, allows to browse through the directories:

```
df <- read.csv(file.choose())
```

Notes

- Unlike `read.table`, `read.csv` defaults to `header = TRUE`, and uses the first row as column names.
- All these functions will convert strings to `factor` class by default unless either `as.is = TRUE` or `stringsAsFactors = FALSE`.
- The `read.csv2` variant defaults to `sep = ";"` and `dec = ","` for use on data from countries where the comma is used as a decimal point and the semicolon as a field separator.

Importing using packages

The `readr` package's `read_csv` function offers much faster performance, a progress bar for large files, and more popular default options than standard `read.csv`, including `stringsAsFactors = FALSE`.

```
library(readr)

df <- read_csv(csv_path)

df
## # A tibble: 4 x 2
##    Var1  Var2
##   <dbl> <chr>
## 1  2.70     A
## 2  3.14     B
## 3 10.00     A
## 4 -7.00     A
```

Importing with data.table

The `data.table` package introduces the function `fread`. While it is similar to `read.table`, `fread` is usually faster and more flexible, guessing the file's delimiter automatically.

```
# get the file path of a CSV included in R's utils package
csv_path <- system.file("misc", "exDIF.csv", package = "utils")

# path will vary based on R installation location
csv_path
## [1] "/Library/Frameworks/R.framework/Resources/library/utils/misc/exDIF.csv"

dt <- fread(csv_path)

dt
##     Var1 Var2
## 1:  2.70    A
## 2:  3.14    B
## 3: 10.00    A
## 4: -7.00    A
```

Where argument `input` is a string representing:

- the filename (*e.g.* `"filename.csv"`),
- a shell command that acts on a file (*e.g.* `"grep 'word' filename"`), or
- the input itself (*e.g.* `"input1, input2 \n A, B \n C, D"`).

`fread` returns an object of class `data.table` that inherits from class `data.frame`, suitable for use with the data.table's usage of `[]`. To return an ordinary data.frame, set the `data.table` parameter to `FALSE`:

```
df <- fread(csv_path, data.table = FALSE)

class(df)
## [1] "data.frame"

df
##    Var1 Var2
```


www.dbooks.org

```
## 1  2.70    A
## 2  3.14    B
## 3 10.00    A
## 4 -7.00    A
```

Notes

- `fread` does not have all same options as `read.table`. One missing argument is `na.comment`, which may lead in unwanted behaviors if the source file contains `#`.
- `fread` uses only `"` for `quote` parameter.
- `fread` uses few (5) lines to guess variables types.

Importing .tsv files as matrices (basic R)

Many people don't make use of `file.path` when making path to a file. But if you are working across Windows, Mac and Linux machines it's usually good practice to use it for making paths instead of `paste`.

```
FilePath <- file.path(AVariableWithFullProjectPath,"SomeSubfolder","SomeFileName.txt.gz")

Data <- as.matrix(read.table(FilePath, header=FALSE, sep ="\t"))
```

Generally this is sufficient for most people.

Sometimes it happens the matrix dimensions are so large that procedure of memory allocation must be taken into account while reading in the matrix, which means reading in the matrix line by line.

Take the previous example, In this case `FilePath` contains a file of dimension `8970 8970` with 79% of the cells containing non-zero values.

```
system.time(expr=Data<-as.matrix(read.table(file=FilePath,header=FALSE,sep=" ") ))
```

`system.time` says 267 seconds were taken to read the file.

```
   user  system elapsed
265.563   1.949 267.563
```

Similarly this file can be read line by line,

```
FilePath <- "SomeFile"
connection<- gzfile(FilePath,open="r")
TableList <- list()
Counter <- 1
system.time(expr= while ( length( Vector<-as.matrix(scan(file=connection, sep=" ", nlines=1,
quiet=TRUE)) ) > 0 ) {
    TableList[[Counter]]<-Vector
    Counter<-Counter+1
})
   user  system elapsed
```

```
165.976   0.060 165.941
close(connection)
system.time(expr=(Data <- do.call(rbind,TableList)))
   user  system elapsed
  0.477   0.088   0.565
```

There's also the `futile.matrix` package which implements a `read.matrix` method, the code itself will reveal itself to be the same thing as described in example 1.

Exporting .csv files

Exporting using base R

Data can be written to a CSV file using `write.csv()`:

```
write.csv(mtcars, "mtcars.csv")
```

Commonly-specified parameters include `row.names = FALSE` and `na = ""`.

Exporting using packages

`readr::write_csv` is significantly faster than `write.csv` and does not write row names.

```
library(readr)

write_csv(mtcars, "mtcars.csv")
```

Import multiple csv files

```
files = list.files(pattern="*.csv")
data_list = lapply(files, read.table, header = TRUE)
```

This read every file and adds it to a list. Afterwards, if all data.frame have the same structure they can be combined into one big data.frame:

```
df <- do.call(rbind, data_list)
```

Importing fixed-width files

Fixed-width files are text files in which columns are not separated by any character delimiter, like `,` or `;`, but rather have a fixed character length (*width*). Data is usually padded with white spaces.

An example:

www.dbooks.org

```
Column1 Column2   Column3           Column4Column5
1647    pi        'important'       3.141596.28318
1731    euler     'quite important' 2.718285.43656
1979    answer    'The Answer.'     42     42
```

Let's assume this data table exists in the local file `constants.txt` in the working directory.

Importing with base R

```
df <- read.fwf('constants.txt', widths = c(8,10,18,7,8), header = FALSE, skip = 1)

df
#>     V1     V2                 V3         V4        V5
#> 1 1647     pi          'important'   3.14159   6.28318
#> 2 1731  euler   'quite important'   2.71828   5.43656
#> 3 1979 answer       'The Answer.'   42        42.0000
```

Note:

- Column titles don't need to be separated by a character (`Column4Column5`)
- The `widths` parameter defines the width of each column
- Non-separated headers are not readable with `read.fwf()`

Importing with readr

```
library(readr)

df <- read_fwf('constants.txt',
               fwf_cols(Year = 8, Name = 10, Importance = 18, Value = 7, Doubled = 8),
               skip = 1)
df
#> # A tibble: 3 x 5
#>    Year    Name        Importance    Value  Doubled
#>    <int>   <chr>           <chr>     <dbl>    <dbl>
#> 1  1647      pi       'important'  3.14159  6.28318
#> 2  1731   euler 'quite important'  2.71828  5.43656
#> 3  1979  answer     'The Answer.' 42.00000 42.00000
```

Note:

- readr's `fwf_*` helper functions offer alternative ways of specifying column lengths, including automatic guessing (`fwf_empty`)
- readr is faster than base R
- Column titles cannot be automatically imported from data file

Read Reading and writing tabular data in plain-text files (CSV, TSV, etc.) online:
https://riptutorial.com/r/topic/481/reading-and-writing-tabular-data-in-plain-text-files--csv--tsv--etc--

Chapter 92: Recycling

Remarks

What is recycling in R

Recycling is when an object is automatically extended in certain operations to match the length of another, longer object.

For example, the vectorised addition results in the following:

```
c(1,2,3) + c(1,2,3,4,5,6)
[1] 2 4 6 5 7 9
```

Because of the recycling, the operation that actually happened was:

```
c(1,2,3,1,2,3) + c(1,2,3,4,5,6)
```

In cases where the longer object is not a multiple of the shorter one, a warning message is presented:

```
c(1,2,3) + c(1,2,3,4,5,6,7)
[1] 2 4 6 5 7 9 8
Warning message:
In c(1, 2, 3) + c(1, 2, 3, 4, 5, 6, 7) :
  longer object length is not a multiple of shorter object length
```

Another example of recycling:

```
matrix(nrow =5, ncol = 2, 1:5 )
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    4
[5,]    5    5
```

Examples

Recycling use in subsetting

Recycling can be used in a clever way to simplify code.

Subsetting

If we want to keep every third element of a vector we can do the following:

www.dbooks.org

```
my_vec <-   c(1,2,3,4,5,6,7,8,9,10)
my_vec[c(TRUE, FALSE)]

[1] 1 3 5 7 9
```

Here the logical expression was expanded to the length of the vector.

We can also perform comparisons using recycling:

```
my_vec <-   c("foo", "bar", "soap", "mix")
my_vec == "bar"

[1] FALSE  TRUE FALSE FALSE
```

Here "bar" gets recycled.

Read Recycling online: https://riptutorial.com/r/topic/5649/recycling

Chapter 93: Regular Expression Syntax in R

Introduction

This document introduces the basics of regular expressions as used in R. For more information about R's regular expression syntax, see `?regex`. For a comprehensive list of regular expression operators, see this ICU guide on regular expressions.

Examples

Use `grep` to find a string in a character vector

```
# General syntax:
# grep(<pattern>, <character vector>)

mystring <- c('The number 5',
              'The number 8',
              '1 is the loneliest number',
              'Company, 3 is',
              'Git SSH tag is git@github.com',
              'My personal site is www.personal.org',
              'path/to/my/file')

grep('5', mystring)
# [1] 1
grep('@', mystring)
# [1] 5
grep('number', mystring)
# [1] 1 2 3
```

`x|y` means look for "x" or "y"

```
grep('5|8', mystring)
# [1] 1 2
grep('com|org', mystring)
# [1] 5 6
```

. is a special character in Regex. It means "match any character"

```
grep('The number .', mystring)
# [1] 1 2
```

Be careful when trying to match dots!

```
tricky <- c('www.personal.org', 'My friend is a cyborg')
grep('.org', tricky)
# [1] 1 2
```

To match a literal character, you have to escape the string with a backslash (`\`). However, R tries

www.dbooks.org

to look for escape characters when creating strings, so you actually need to escape the backslash itself (i.e. you need to *double escape* regular expression characters.)

```
grep('\.org', tricky)
# Error: '\.' is an unrecognized escape in character string starting "'\."
grep('\\.org', tricky)
# [1] 1
```

If you want to match one of several characters, you can wrap those characters in brackets (`[]`)

```
grep('[13]', mystring)
# [1] 3 4
grep('[@/]', mystring)
# [1] 5 7
```

It may be useful to indicate character sequences. E.g. `[0-4]` will match 0, 1, 2, 3, or 4, `[A-Z]` will match any uppercase letter, `[A-z]` will match any uppercase or lowercase letter, and `[A-z0-9]` will match any letter or number (i.e. all alphanumeric characters)

```
grep('[0-4]', mystring)
# [1] 3 4
grep('[A-Z]', mystring)
# [1] 1 2 4 5 6
```

R also has several shortcut classes that can be used in brackets. For instance, `[:lower:]` is short for `a-z`, `[:upper:]` is short for `A-Z`, `[:alpha:]` is `A-z`, `[:digit:]` is `0-9`, and `[:alnum:]` is `A-z0-9`. Note that these *whole expressions* must be used inside brackets; for instance, to match a single digit, you can use `[[:digit:]]` (note the double brackets). As another example, `[@[:digit:]/]` will match the characters `@`, `/` or `0-9`.

```
grep('[[:digit:]]', mystring)
# [1] 1 2 3 4
grep('[@[:digit:]/]', mystring)
# [1] 1 2 3 4 5 7
```

Brackets can also be used to negate a match with a carat (^). For instance, `[^5]` will match any character other than "5".

```
grep('The number [^5]', mystring)
# [1] 2
```

Read Regular Expression Syntax in R online: https://riptutorial.com/r/topic/9743/regular-expression-syntax-in-r

Chapter 94: Regular Expressions (regex)

Introduction

Regular expressions (also called "regex" or "regexp") define patterns that can be matched against a string. Type `?regex` for the official R documentation and see the Regex Docs for more details. The most important 'gotcha' that will not be learned in the SO regex/topics is that most R-regex functions need the use of paired backslashes to escape in a `pattern` parameter.

Remarks

Character classes

- `"[AB]"` could be A or B
- `"[[:alpha:]]"` could be any letter
- `"[[:lower:]]"` stands for any lower-case letter. Note that `"[a-z]"` is close but doesn't match, e.g., `ú`.
- `"[[:upper:]]"` stands for any upper-case letter. Note that `"[A-Z]"` is close but doesn't match, e.g., `Ú`.
- `"[[:digit:]]"` stands for any digit : 0, 1, 2, ..., or 9 and is equivalent to `"[0-9]"`.

Quantifiers

`+`, `*` and `?` apply as usual in regex. -- `+` matches at least once, `*` matches 0 or more times, and `?` matches 0 or 1 time.

Start and end of line indicators

You can specify the position of the regex in the string :

- `"^..."` forces the regular expression to be at the beginning of the string
- `"...$"` forces the regular expression to be at the end of the string

Differences from other languages

Please note that regular expressions in R often look *ever-so-slightly* different from regular expressions used in other languages.

- R requires double-backslash escapes (because `"\"` already implies escaping in general in R strings), so, for example, to capture whitespace in most regular expression engines, one

www.dbooks.org

simply needs to type `\s`, vs. `\\s` in R.

- UTF-8 characters in R should be escaped with a capital U, e.g. `[\U{1F600}]` and `[\U1F600]` match , whereas in, e.g., Ruby, this would be matched with a lower-case u.

Additional Resources

The following site reg101 is a good place for checking online regex before using it R-script.

The R Programmming wikibook has a page dedicated to text processing with many examples using regular expressions.

Examples

Eliminating Whitespace

```
string <- '    some text on line one;
and then some text on line two     '
```

Trimming Whitespace

"Trimming" whitespace typically refers to removing both leading and trailing whitespace from a string. This may be done using a combination of the previous examples. `gsub` is used to force the replacement over both the leading and trailing matches.

Prior to R 3.2.0

```
gsub(pattern = "(^ +| +$)",
     replacement = "",
     x = string)

[1] "some text on line one; \nand then some text on line two"
```

R 3.2.0 and higher

```
trimws(x = string)

[1] "some text on line one; \nand then some text on line two"
```

Removing Leading Whitespace

Prior to R 3.2.0

```
sub(pattern = "^ +",
    replacement = "",
    x = string)
```

```
[1] "some text on line one; \nand then some text on line two     "
```

R 3.2.0 and higher

```
trimws(x = string,
       which = "left")

[1] "some text on line one; \nand then some text on line two     "
```

Removing Trailing Whitespace

Prior to R 3.2.0

```
sub(pattern = " +$",
    replacement = "",
    x = string)

[1] "    some text on line one; \nand then some text on line two"
```

R 3.2.0 and higher

```
trimws(x = string,
       which = "right")

[1] "    some text on line one; \nand then some text on line two"
```

Removing All Whitespace

```
gsub(pattern = "\\s",
     replacement = "",
     x = string)

[1] "sometextonlineone;andthensometextonlinetwo"
```

Note that this will also remove white space characterse such as tabs (`\t`), newlines (`\r` and `\n`), and spaces.

Validate a date in a "YYYYMMDD" format

It is a common practice to name files using the date as prefix in the following format: `YYYYMMDD`, for example: `20170101_results.csv`. A date in such string format can be verified using the following regular expression:

```
\\d{4}(0[1-9]|1[012])(0[1-9]|[12][0-9]|3[01])
```

The above expression considers dates from year: `0000-9999`, months between: `01-12` and days `01-31`.

www.dbooks.org

For example:

```
> grepl("\\d{4}(0[1-9]|1[012])(0[1-9]|[12][0-9]|3[01])", "20170101")
[1] TRUE
> grepl("\\d{4}(0[1-9]|1[012])(0[1-9]|[12][0-9]|3[01])", "20171206")
[1] TRUE
> grepl("\\d{4}(0[1-9]|1[012])(0[1-9]|[12][0-9]|3[01])", "29991231")
[1] TRUE
```

Note: It validates the date syntax, but we can have a wrong date with a valid syntax, for example: `20170229` (2017 it is not a leap year).

```
> grepl("\\d{4}(0[1-9]|1[012])(0[1-9]|[12][0-9]|3[01])", "20170229")
[1] TRUE
```

If you want to validate a date, it can be done via this user defined function:

```
is.Date <- function(x) {return(!is.na(as.Date(as.character(x), format = '%Y%m%d')))}
```

Then

```
> is.Date(c("20170229", "20170101", 20170101))
[1] FALSE  TRUE  TRUE
```

Validate US States postal abbreviations

The following `regex` includes 50 states and also Commonwealth/Territory (see www.50states.com):

```
regex <-
"(A[LKSZR])|(C[AOT])|(D[EC])|(F[ML])|(G[AU])|(HI)|(I[DLNA])|(K[SY])|(LA)|(M[EHDAINSOT])|(N[EVHJMYCD])|(M
```

For example:

```
> test <- c("AL", "AZ", "AR", "AJ", "AS", "DC", "FM", "GU","PW", "FL", "AJ", "AP")
> grepl(us.states.pattern, test)
 [1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
>
```

Note:

If you want to verify only the 50 States, then we recommend to use the R-dataset: `state.abb` from `state`, for example:

```
> data(state)
> test %in% state.abb
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

We get `TRUE` only for 50-States abbreviations: `AL, AZ, AR, FL`.

Validate US phone numbers

The following regular expression:

```
us.phones.regex <- "^\\s*(\\+\\s*1(-?|\\s+))*[0-9]{3}\\s*-?\\s*[0-9]{3}\\s*-?\\s*[0-9]{4}$"
```

Validates a phone number in the form of: `+1-xxx-xxx-xxxx`, including optional leading/trailing blanks at the beginning/end of each group of numbers, but not in the middle, for example: `+1-xxx-xxx-xx xx` is not valid. The `-` delimiter can be replaced by blanks: `xxx xxx xxx` or without delimiter: `xxxxxxxxxx`. The `+1` prefix is optional.

Let's check it:

```
us.phones.regex <- "^\\s*(\\+\\s*1(-?|\\s+))*[0-9]{3}\\s*-?\\s*[0-9]{3}\\s*-?\\s*[0-9]{4}$"

phones.OK <- c("305-123-4567", "305 123 4567", "+1-786-123-4567",
    "+1 786 123 4567", "7861234567", "786 - 123   4567", "+ 1 786 - 123   4567")

phones.NOK <- c("124-456-78901", "124-456-789", "124-456-78 90",
    "124-45 6-7890", "12 4-456-7890")
```

Valid cases:

```
> grepl(us.phones.regex, phones.OK)
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
>
```

Invalid cases:

```
 > grepl(us.phones.regex, phones.NOK)
[1] FALSE FALSE FALSE FALSE FALSE
>
```

Note:

- `\\s` Matches any space, tab or newline character

Escaping characters in R regex patterns

Since both R and regex share the escape character ,`"\"`, building correct patterns for `grep`, `sub`, `gsub` or any other function that accepts a pattern argument will often need pairing of backslashes. If you build a three item character vector in which one items has a linefeed, another a tab character and one neither, and hte desire is to turn either the linefeed or the tab into 4-spaces then a single backslash is need for the construction, but tpaired backslashes for matching:

```
x <- c( "a\nb", "c\td", "e    f")
x  # how it's stored
   #  [1] "a\nb"   "c\td"   "e    f"
cat(x)   # how it will be seen with cat
#a
#b c    d e    f

gsub(patt="\\n|\\t", repl="    ", x)
```


www.dbooks.org

```
#[1] "a    b" "c    d" "e    f"
```

Note that the pattern argument (which is optional if it appears first and only needs partial spelling) is the only argument to require this doubling or pairing. The replacement argument does not require the doubling of characters needing to be escaped. If you wanted all the linefeeds and 4-space occurrences replaces with tabs it would be:

```
gsub("\\n|    ", "\t", x)
#[1] "a\tb" "c\td" "e\tf"
```

Differences between Perl and POSIX regex

There are two ever-so-slightly different engines of regular expressions implemented in R. The default is called POSIX-consistent; all regex functions in R are also equipped with an option to turn on the latter type: `perl = TRUE`.

Look-ahead/look-behind

`perl = TRUE` enables look-ahead and look-behind in regular expressions.

- `"(?<=A)B"` matches an appearance of the letter `B` *only if* it's preceded by `A`, i.e. `"ABACADABRA"` would be matched, but `"abacadabra"` and `"aBacadabra"` would not.

Read Regular Expressions (regex) online: https://riptutorial.com/r/topic/5748/regular-expressions--regex-

Chapter 95: Reproducible R

Introduction

With 'Reproducibility' we mean that someone else (perhaps you in the future) can repeat the steps you performed and get the same result. See the Reproducible Research Task View.

Remarks

To create reproducible results, all sources of variation need to be fixed. For instance, if a (pseudo)random number generator is used, the seed needs to be fixed if you want to recreate the same results. Another way to reduce variation is to combine text and computation in the same document.

References

- Peng, R. D. (2011). Reproducible Research in Computational. Science, 334(6060), 1226–1227. http://doi.org/10.1126/science.1213847
- Peng, Roger D. Report Writing for Data Science in R. Leanpub, 2015. https://leanpub.com/reportwriting.

Examples

Data reproducibility

`dput()` and `dget()`

The easiest way to share a (preferable small) data frame is to use a basic function `dput()`. It will export an R object in a plain text form.

Note: Before making the example data below, make sure you're in an empty folder you can write to. Run `getwd()` and read `?setwd` if you need to change folders.

```
dput(mtcars, file = 'df.txt')
```

Then, anyone can load the precise R object to their GlobalEnvironment using the `dget()` function.

```
df <- dget('df.txt')
```

For larger R objects, there are a number of ways of saving them reproducibly. See Input and output .

Package reproducibility

Package reproducibility is a very common issue in reproducing some R code. When various packages get updated, some interconnections between them may break. The ideal solution for the problem is to reproduce the image of the R code writer's machine on your computer at the date when the code was written. And here comes `checkpoint` package.

Starting from 2014-09-17, the authors of the package make daily copies of the whole CRAN package repository to their own mirror repository -- Microsoft R Archived Network. So, to avoid package reproduciblity issues when creating a reproducible R project, all you need is to:

1. Make sure that all your packages (and R version) are up-to-date.
2. Include `checkpoint::checkpoint('YYYY-MM-DD')` line in your code.

`checkpoint` will create a directory `.checkpoint` in your R_home directory (`"~/"`). To this technical directory it will install all the packages, that are used in your project. That means, `checkpoint` looks through all the `.R` files in your project directory to pick up all the `library()` or `require()` calls and install all the required packages in the form they existed at CRAN on the specified date.

PRO You are freed from the package reproducibility issue.
CONTRA For each specified date you have to download and install all the packages that are used in a certain project that you aim to reproduce. That may take quite a while.

Read Reproducible R online: https://riptutorial.com/r/topic/4087/reproducible-r

Chapter 96: Reshape using tidyr

Introduction

tidyr has two tools for reshaping data: `gather` (wide to long) and `spread` (long to wide).

See Reshaping data for other options.

Examples

Reshape from long to wide format with spread()

```
library(tidyr)

## example data
set.seed(123)
df <- data.frame(
  name = rep(c("firstName", "secondName"), each=4),
  numbers = rep(1:4, 2),
  value = rnorm(8)
)
df
#          name numbers       value
# 1   firstName       1 -0.56047565
# 2   firstName       2 -0.23017749
# 3   firstName       3  1.55870831
# 4   firstName       4  0.07050839
# 5  secondName       1  0.12928774
# 6  secondName       2  1.71506499
# 7  secondName       3  0.46091621
# 8  secondName       4 -1.26506123
```

We can "spread" the 'numbers' column, into separate columns:

```
spread(data = df,
       key = numbers,
       value = value)
#       name          1          2         3           4
# 1   firstName -0.5604756 -0.2301775 1.5587083  0.07050839
# 2  secondName  0.1292877  1.7150650 0.4609162 -1.26506123
```

Or spread the 'name' column into separate columns:

```
spread(data = df,
       key = name,
       value = value)
#   numbers   firstName secondName
# 1       1 -0.56047565  0.1292877
# 2       2 -0.23017749  1.7150650
# 3       3  1.55870831  0.4609162
# 4       4  0.07050839 -1.2650612
```


www.dbooks.org

Reshape from wide to long format with gather()

```
library(tidyr)

## example data
df <- read.table(text ="  numbers  firstName secondName
1       1  1.5862639  0.4087477
2       2  0.1499581  0.9963923
3       3  0.4117353  0.3740009
4       4 -0.4926862  0.4437916", header = T)
df
#   numbers  firstName secondName
# 1       1  1.5862639  0.4087477
# 2       2  0.1499581  0.9963923
# 3       3  0.4117353  0.3740009
# 4       4 -0.4926862  0.4437916
```

We can gather the columns together using 'numbers' as the key column:

```
gather(data = df,
       key = numbers,
       value = myValue)
#   numbers    numbers    myValue
# 1       1  firstName  1.5862639
# 2       2  firstName  0.1499581
# 3       3  firstName  0.4117353
# 4       4  firstName -0.4926862
# 5       1 secondName  0.4087477
# 6       2 secondName  0.9963923
# 7       3 secondName  0.3740009
# 8       4 secondName  0.4437916
```

Read Reshape using tidyr online: https://riptutorial.com/r/topic/9195/reshape-using-tidyr

Chapter 97: Reshaping data between long and wide forms

Introduction

In R, tabular data is stored in data frames. This topic covers the various ways of transforming a single table.

Remarks

Helpful packages

- Reshaping, stacking and splitting with data.table
- Reshape using tidyr
- splitstackshape

Examples

The reshape function

The most flexible base R function for reshaping data is `reshape`. See `?reshape` for its syntax.

```
# create unbalanced longitudinal (panel) data set
set.seed(1234)
df <- data.frame(identifier=rep(1:5, each=3),
                 location=rep(c("up", "down", "left", "up", "center"), each=3),
                 period=rep(1:3, 5), counts=sample(35, 15, replace=TRUE),
                 values=runif(15, 5, 10))[-c(4,8,11),]
df

   identifier location period counts   values
1           1       up      1      4 9.186478
2           1       up      2     22 6.431116
3           1       up      3     22 6.334104
5           2     down      2     31 6.161130
6           2     down      3     23 6.583062
7           3     left      1      1 6.513467
9           3     left      3     24 5.199980
10          4       up      1     18 6.093998
12          4       up      3     20 7.628488
13          5   center      1     10 9.573291
14          5   center      2     33 9.156725
15          5   center      3     11 5.228851
```

Note that the data.frame is unbalanced, that is, unit 2 is missing an observation in the first period, while units 3 and 4 are missing observations in the second period. Also, note that there are two variables that vary over the periods: counts and values, and two that do not vary: identifier and

www.dbooks.org

location.

Long to Wide

To reshape the data.frame to wide format,

```
# reshape wide on time variable
df.wide <- reshape(df, idvar="identifier", timevar="period",
                   v.names=c("values", "counts"), direction="wide")
df.wide
   identifier location values.1 counts.1 values.2 counts.2 values.3 counts.3
1           1       up 9.186478        4 6.431116       22 6.334104       22
5           2     down       NA       NA 6.161130       31 6.583062       23
7           3     left 6.513467        1       NA       NA 5.199980       24
10          4       up 6.093998       18       NA       NA 7.628488       20
13          5   center 9.573291       10 9.156725       33 5.228851       11
```

Notice that the missing time periods are filled in with NAs.

In reshaping wide, the "v.names" argument specifies the columns that vary over time. If the location variable is not necessary, it can be dropped prior to reshaping with the "drop" argument. In dropping the only non-varying / non-id column from the data.frame, the v.names argument becomes unnecessary.

```
reshape(df, idvar="identifier", timevar="period", direction="wide",
        drop="location")
```

Wide to Long

To reshape long with the current df.wide, a minimal syntax is

```
reshape(df.wide, direction="long")
```

However, this is typically trickier:

```
# remove "." separator in df.wide names for counts and values
names(df.wide)[grep("\\.", names(df.wide))] <-
              gsub("\\.", "", names(df.wide)[grep("\\.", names(df.wide))])
```

Now the simple syntax will produce an error about undefined columns.

With column names that are more difficult for the `reshape` function to automatically parse, it is sometimes necessary to add the "varying" argument which tells `reshape` to group particular variables in wide format for the transformation into long format. This argument takes a list of vectors of variable names or indices.

```
reshape(df.wide, idvar="identifier",
        varying=list(c(3,5,7), c(4,6,8)), direction="long")
```

In reshaping long, the "v.names" argument can be provided to rename the resulting varying variables.

Sometimes the specification of "varying" can be avoided by use of the "sep" argument which tells `reshape` what part of the variable name specifies the value argument and which specifies the time argument.

Reshaping data

Often data comes in tables. Generally one can divide this tabular data in wide and long formats. In a wide format, each variable has its own column.

Person	Height [cm]	Age [yr]
Alison	178	20
Bob	174	45
Carl	182	31

However, sometimes it is more convenient to have a long format, in which all variables are in one column and the values are in a second column.

Person	Variable	Value
Alison	Height [cm]	178
Bob	Height [cm]	174
Carl	Height [cm]	182
Alison	Age [yr]	20
Bob	Age [yr]	45
Carl	Age [yr]	31

Base R, as well as third party packages can be used to simplify this process. For each of the options, the `mtcars` dataset will be used. By default, this dataset is in a long format. In order for the packages to work, we will insert the row names as the first column.

```
mtcars # shows the dataset
data <- data.frame(observation=row.names(mtcars),mtcars)
```

Base R

There are two functions in base R that can be used to convert between wide and long format:

www.dbooks.org

`stack()` and `unstack()`.

```
long <- stack(data)
long # this shows the long format
wide <- unstack(long)
wide # this shows the wide format
```

However, these functions can become very complex for more advanced use cases. Luckily, there are other options using third party packages.

The tidyr package

This package uses `gather()` to convert from wide to long and `spread()` to convert from long to wide.

```
library(tidyr)
long <- gather(data, variable, value, 2:12) # where variable is the name of the
# variable column, value indicates the name of the value column and 2:12 refers to
# the columns to be converted.
long # shows the long result
wide <- spread(long,variable,value)
wide # shows the wide result (~data)
```

The data.table package

The data.table package extends the `reshape2` functions and uses the function `melt()` to go from wide to long and `dcast()` to go from long to wide.

```
library(data.table)
long <- melt(data,'observation',2:12,'variable', 'value')
long # shows the long result
wide <- dcast(long, observation ~ variable)
wide # shows the wide result (~data)
```

Read Reshaping data between long and wide forms online:
https://riptutorial.com/r/topic/2904/reshaping-data-between-long-and-wide-forms

Chapter 98: RESTful R Services

Introduction

OpenCPU uses standard R packaging to develop, ship and deploy web applications.

Examples

opencpu Apps

The official website contain good exemple of apps: https://www.opencpu.org/apps.html

The following code is used to serve a R session:

```
library(opencpu)
opencpu$start(port = 5936)
```

After this code is executed, you can use URLs to access the functions of the R session. The result could be XML, html, JSON or some other defined formats.

For exemple, the previous R session can be accessed by a cURL call:

```
#curl uses http post method for -X POST or -d "arg=value"
curl http://localhost:5936/ocpu/library/MASS/scripts/ch01.R -X POST
curl http://localhost:5936/ocpu/library/stats/R/rnorm -d "n=10&mean=5"
```

The call is asynchronous, meaning that the R session is not blocked while waiting for the call to finish (contrary to shiny).

The call result is kept in a temporary session stored in `/ocpu/tmp/`

An exemple of how to retrieve the temporary session:

```
curl https://public.opencpu.org/ocpu/library/stats/R/rnorm -d n=5
/ocpu/tmp/x009f9e7630/R/.val
/ocpu/tmp/x009f9e7630/stdout
/ocpu/tmp/x009f9e7630/source
/ocpu/tmp/x009f9e7630/console
/ocpu/tmp/x009f9e7630/info
```

`x009f9e7630` is the name of the session.

Pointing to `/ocpu/tmp/x009f9e7630/R/.val` will return the value resulting of `rnorm(5)`, `/ocpu/tmp/x009f9e7630/R/console` will return the content of the console of `rnorm(5)`, etc..

Read RESTful R Services online: https://riptutorial.com/r/topic/8323/restful-r-services

www.dbooks.org

Chapter 99: RMarkdown and knitr presentation

Syntax

- Header:
  - YAML format, used when the script is compile to define general parameter and metadata

Parameters

Parameter	definition
title	the title of the document
author	The author of the document
date	The date of the document: Can be "`r format(Sys.time(), '%d %B, %Y')`"
author	The author of the document
output	The output format of the document: at least 10 format available. For html document, `html_output`. For PDF document, `pdf_document`, ..

Remarks

Sub options parameters:

sub-option	description	html	pdf	word	odt	rtf	md	github	ioslides	slidy
citation_package	The LaTeX package to process citations, natbib, biblatex or none		X				X			
code_folding	Let readers to toggle the display of R code, "none", "hide", or "show"	X								

sub-option	description	html	pdf	word	odt	rtf	md	github	ioslides	slidy
colortheme	Beamer color theme to use									
css	CSS file to use to style document	X							X	X
dev	Graphics device to use for figure output (e.g. "png")	X	X				X	X	X	X
duration	Add a countdown timer (in minutes) to footer of slides									X
fig_caption	Should figures be rendered with captions?	X	X	X	X				X	X
fig_height, fig_width	Default figure height and width (in inches) for document	X	X	X	X	X	X	X	X	X
highlight	Syntax highlighting: "tango", "pygments", "kate","zenburn", "textmate"	X	X	X						X
includes	File of content to place in document (in_header, before_body, after_body)	X	X		X		X	X	X	X
incremental	Should bullets appear one at a time (on presenter mouse clicks)?								X	X
keep_md	Save a copy of	X		X	X	X			X	X

www.dbooks.org

sub-option	description	html	pdf	word	odt	rtf	md	github	ioslides	slidy
	.md file that contains knitr output									
keep_tex	Save a copy of .tex file that contains knitr output		X							
latex_engine	Engine to render latex, or ""pdflatex", "xelatex", lualatex"		X							
lib_dir	Directory of dependency files to use (Bootstrap, MathJax, etc.)	X							X	X
mathjax	Set to local or a URL to use a local/URL version of MathJax to render	X							X	X
md_extensions	Markdown extensions to add to default definition or R Markdown	X	X	X	X	X	X	X	X	X
number_sections	Add section numbering to headers	X	X							
pandoc_args	Additional arguments to pass to Pandoc	X	X	X	X	X	X	X	X	X
preserve_yaml	Preserve YAML front matter in final document?						X			

sub-option	description	html	pdf	word	odt	rtf	md	github	ioslides	slidy
reference_docx	docx file whose styles should be copied when producing docx output			X						
self_contained	Embed dependencies into the doc	X							X	X
slide_level	The lowest heading level that defines individual slides									
smaller	Use the smaller font size in the presentation?								X	
smart	Convert straight quotes to curly, dashes to em-dashes, ... to ellipses, etc.	X							X	X
template	Pandoc template to use when rendering file	X	X		X					X
theme	Bootswatch or Beamer theme to use for page	X								
toc	Add a table of contents at start of document	X	X	X		X	X	X		
toc_depth	The lowest level of headings to add to table of contents	X	X	X		X	X	X		
toc_float	Float the table of contents to the left of the main	X								

www.dbooks.org

sub-option	description	html	pdf	word	odt	rtf	md	github	ioslides	slidy
	content									

Examples

Rstudio example

This is a script saved as .Rmd, on the contrary of r scripts saved as .R.

To knit the script, either use the `render` function or use the shortcut button in Rstudio.

````
---
title: "Rstudio exemple of a rmd file"
author: 'stack user'
date: "22 July 2016"
output: html_document
---

The header is used to define the general parameters and the metadata.

## R Markdown

This is an R Markdown document.
It is a script written in markdown with the possibility to insert chunk of R code in it.
To insert R code, it needs to be encapsulated into inverted quote.

Like that for a long piece of code:

```{r cars}
summary(cars)
```

And like ``r cat("that")`` for small piece of code.

## Including Plots

You can also embed plots, for example:

```{r echo=FALSE}
plot(pressure)
```
````

Adding a footer to an ioslides presentation

Adding a footer is not natively possible. Luckily, we can make use of jQuery and CSS to add a footer to the slides of an ioslides presentation rendered with knitr. First of all we have to include the jQuery plugin. This is done by the line

```
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.2/jquery.min.js"></script>
```

Now we can use jQuery to alter the DOM (*document object model*) of our presentation. In other words: we alter the HTML structure of the document. As soon as the presentation is loaded (`$(document).ready(function() { ... })`

), we select all slides, that do not have the class attributes `.title-slide`, `.backdrop`, or `.segue` and add the tag `<footer></footer>` right before each slide is 'closed' (so before `</slide>`). The attribute `label` carries the content that will be displayed later on.

All we have to do now is to layout our footer with CSS:

After each `<footer>` (`footer::after`):

- display the content of the attribute `label`
- use font size 12
- position the footer (20 pixels from the bottom of the slide and 60 pxs from the left)

(the other properties can be ignored but might have to be modified if the presentation uses a different style template).

```
---
title: "Adding a footer to presentaion slides"
author: "Martin Schmelzer"
date: "26 Juli 2016"
output: ioslides_presentation
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE)
```

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.2/jquery.min.js"></script>

<script>
    $(document).ready(function() {
      $('slide:not(.title-slide, .backdrop, .segue)').append('<footer label=\"My amazing
footer!\"></footer>');
    })
</script>

<style>
  footer:after {
    content: attr(label);
    font-size: 12pt;
    position: absolute;
    bottom: 20px;
    left: 60px;
    line-height: 1.9;
  }
</style>

Slide 1

This is slide 1.

Slide 2

This is slide 2

Test

Slide 3
```

www.dbooks.org

```
And slide 3.
```

The result will look like this:



Read RMarkdown and knitr presentation online: https://riptutorial.com/r/topic/2999/rmarkdown-and-knitr-presentation

Chapter 100: RODBC

Examples

Connecting to Excel Files via RODBC

While `RODBC` is restricted to Windows computers with compatible architecture between R and any target RDMS, one of its key flexibilities is to work with Excel files as if they were SQL databases.

```
require(RODBC)
con = odbcConnectExcel("myfile.xlsx") # open a connection to the Excel file
sqlTables(con)$TABLE_NAME # show all sheets
df = sqlFetch(con, "Sheet1") # read a sheet
df = sqlQuery(con, "select * from [Sheet1 $]") # read a sheet (alternative SQL syntax)
close(con) # close the connection to the file
```

SQL Server Management Database connection to get individual table

Another use of RODBC is in connecting with SQL Server Management Database. We need to specify the 'Driver' i.e. SQL Server here, the database name "Atilla" and then use the `sqlQuery` to extract either the full table or a fraction of it.

```
library(RODBC)
cn  <- odbcDriverConnect(connection="Driver={SQL
Server};server=localhost;database=Atilla;trusted_connection=yes;")
tbl <- sqlQuery(cn, 'select top 10 * from table_1')
```

Connecting to relational databases

```
library(RODBC)
con <- odbcDriverConnect("driver={Sql Server};server=servername;trusted connection=true")
dat <- sqlQuery(con, "select * from table");
close(con)
```

This will connect to a SQL Server instance. For more information on what your connection string should look like, visit connectionstrings.com

Also, since there's no database specified, you should make sure you fully qualify the object you're wanting to query like this databasename.schema.objectname

Read RODBC online: https://riptutorial.com/r/topic/2471/rodbc

www.dbooks.org

Chapter 101: roxygen2

Parameters

Parameter	details
author	Author of the package
examples	The following lines will be examples on how to use the documented function
export	To export the function - i.e. make it callable by users of the package
import	Package(s) namespace(s) to import
importFrom	Functions to import from the package (first name of the list)
param	Parameter of the function to document

Examples

Documenting a package with roxygen2

Writing with roxygen2

roxygen2 is a package created by Hadley Wickham to facilitate documentation.

It allows to include the documentation inside the R script, in lines starting by `#'`. The different parameters passed to the documentation start with an `@`, for example the creator of a package will by written as follow:

```
#' @author The Author
```

For example, if we wanted to document the following function:

```
mean<-function(x) sum(x)/length(x)
```

We will want to write a small description to this function, and explain the parameters with the following (each line will be explained and detailed after):

```
#' Mean
#'
#' A function to compute the mean of a vector
#' @param x A numeric vector
#' @keyword mean
#' @importFrom base sum
```

```
#' @export
#' @examples
#' mean(1:3)
#' \dontrun{ mean(1:1e99) }
mean<-function(x) sum(x)/length(x)
```

- The first line `#' Mean` is the title of the documentation, the following lines make the corpus.
- Each parameter of a function must be detailed through a relevant `@param`. `@export` indicated that this function name should be exported, and thus can be called when the package is loaded.
- `@keyword` provides relevant keywords when looking for help
- `@importFrom` lists all functions to import from a package that will be used in this function or in you package. Note that importing the complete namespace of a package can be done with `@import`
- The examples are then written below the `@example` tag.
    - The first one will be evaluated when the package is built;
    - The second one will not - usually to prevent long computations - due to the `\dontrun` command.

Building the documentation

The documentation can be created using `devtools::document()`. Note also that `devtools::check()` will automatically create a documentation and will report missing arguments in the documentation of functions as warnings.

Read roxygen2 online: https://riptutorial.com/r/topic/5171/roxygen2

www.dbooks.org

Chapter 102: Run-length encoding

Remarks

A run is a consecutive sequence of repeated values or observations. For repeated values, R's "run-length encoding" concisely describes a vector in terms of its runs. Consider:

```
dat <- c(1, 2, 2, 2, 3, 1, 4, 4, 1, 1)
```

We have a length-one run of 1s; then a length-three run of 2s; then a length-one run of 3s; and so on. R's run-length encoding captures all the lengths and values of a vector's runs.

Extensions

A run can also refer to consecutive observations in a tabular data. While R doesn't have a natural way of encoding these, they can be handled with `rleid` from the data.table package (currently a dead-end link).

Examples

Run-length Encoding with `rle`

Run-length encoding captures the lengths of runs of consecutive elements in a vector. Consider an example vector:

```
dat <- c(1, 2, 2, 2, 3, 1, 4, 4, 1, 1)
```

The `rle` function extracts each run and its length:

```
r <- rle(dat)
r
# Run Length Encoding
#   lengths: int [1:6] 1 3 1 1 2 2
#   values : num [1:6] 1 2 3 1 4 1
```

The values for each run are captured in `r$values`:

```
r$values
# [1] 1 2 3 1 4 1
```

This captures that we first saw a run of 1's, then a run of 2's, then a run of 3's, then a run of 1's, and so on.

The lengths of each run are captured in `r$lengths`:

```
r$lengths
# [1] 1 3 1 1 2 2
```

We see that the initial run of 1's was of length 1, the run of 2's that followed was of length 3, and so on.

Identifying and grouping by runs in base R

One might want to group their data by the runs of a variable and perform some sort of analysis. Consider the following simple dataset:

```
(dat <- data.frame(x = c(1, 1, 2, 2, 2, 1), y = 1:6))
#   x y
# 1 1 1
# 2 1 2
# 3 2 3
# 4 2 4
# 5 2 5
# 6 1 6
```

The variable `x` has three runs: a run of length 2 with value 1, a run of length 3 with value 2, and a run of length 1 with value 1. We might want to compute the mean value of variable `y` in each of the runs of variable `x` (these mean values are 1.5, 4, and 6).

In base R, we would first compute the run-length encoding of the `x` variable using `rle`:

```
(r <- rle(dat$x))
# Run Length Encoding
#   lengths: int [1:3] 2 3 1
#   values : num [1:3] 1 2 1
```

The next step is to compute the run number of each row of our dataset. We know that the total number of runs is `length(r$lengths)`, and the length of each run is `r$lengths`, so we can compute the run number of each of our runs with `rep`:

```
(run.id <- rep(seq_along(r$lengths), r$lengths))
# [1] 1 1 2 2 2 3
```

Now we can use `tapply` to compute the mean `y` value for each run by grouping on the run id:

```
data.frame(x=r$values, meanY=tapply(dat$y, run.id, mean))
#   x meanY
# 1 1   1.5
# 2 2   4.0
# 3 1   6.0
```

Identifying and grouping by runs in data.table

The data.table package provides a convenient way to group by runs in data. Consider the following example data:

www.dbooks.org

```
library(data.table)
(DT <- data.table(x = c(1, 1, 2, 2, 2, 1), y = 1:6))
#    x y
# 1: 1 1
# 2: 1 2
# 3: 2 3
# 4: 2 4
# 5: 2 5
# 6: 1 6
```

The variable `x` has three runs: a run of length 2 with value 1, a run of length 3 with value 2, and a run of length 1 with value 1. We might want to compute the mean value of variable `y` in each of the runs of variable x (these mean values are 1.5, 4, and 6).

The data.table `rleid` function provides an id indicating the run id of each element of a vector:

```
rleid(DT$x)
# [1] 1 1 2 2 2 3
```

One can then easily group on this run ID and summarize the `y` data:

```
DT[,mean(y),by=.(x, rleid(x))]
#    x rleid  V1
# 1: 1     1 1.5
# 2: 2     2 4.0
# 3: 1     3 6.0
```

Run-length encoding to compress and decompress vectors

Long vectors with long runs of the same value can be significantly compressed by storing them in their run-length encoding (the value of each run and the number of times that value is repeated). As an example, consider a vector of length 10 million with a huge number of 1's and only a small number of 0's:

```
set.seed(144)
dat <- sample(rep(0:1, c(1, 1e5)), 1e7, replace=TRUE)
table(dat)
#       0       1
#     103 9999897
```

Storing 10 million entries will require significant space, but we can instead create a data frame with the run-length encoding of this vector:

```
rle.df <- with(rle(dat), data.frame(values, lengths))
dim(rle.df)
# [1] 207   2
head(rle.df)
#   values lengths
# 1      1   52818
# 2      0       1
# 3      1  219329
# 4      0       1
# 5      1  318306
```

```
# 6      0       1
```

From the run-length encoding, we see that the first 52,818 values in the vector are 1's, followed by a single 0, followed by 219,329 consecutive 1's, followed by a 0, and so on. The run-length encoding only has 207 entries, requiring us to store only 414 values instead of 10 million values. As `rle.df` is a data frame, it can be stored using standard functions like `write.csv`.

Decompressing a vector in run-length encoding can be accomplished in two ways. The first method is to simply call `rep`, passing the `values` element of the run-length encoding as the first argument and the `lengths` element of the run-length encoding as the second argument:

```
decompressed <- rep(rle.df$values, rle.df$lengths)
```

We can confirm that our decompressed data is identical to our original data:

```
identical(decompressed, dat)
# [1] TRUE
```

The second method is to use R's built-in `inverse.rle` function on the `rle` object, for instance:

```
rle.obj <- rle(dat)                              # create a rle object here
class(rle.obj)
# [1] "rle"

dat.inv <- inverse.rle(rle.obj)                 # apply the inverse.rle on the rle object
```

We can confirm again that this produces exactly the original `dat`:

```
identical(dat.inv, dat)
# [1] TRUE
```

Read Run-length encoding online: https://riptutorial.com/r/topic/1133/run-length-encoding

www.dbooks.org

Chapter 103: Scope of variables

Remarks

The most common pitfall with scope arises in parallelization. All variables and functions must be passed into a new environment that is run on each thread.

Examples

Environments and Functions

Variables declared inside a function only exist (unless passed) inside that function.

```
x <- 1

foo <- function(x) {
    y <- 3
    z <- x + y
    return(z)
}

y
```

Error: object 'y' not found

Variables passed into a function and then reassigned are overwritten, *but only inside the function*.

```
foo <- function(x) {
    x <- 2
    y <- 3
    z <- x + y
    return(z)
}

foo(1)
x
```

5

1

Variables assigned in a higher environment than a function exist within that function, without being passed.

```
foo <- function() {
    y <- 3
    z <- x + y
    return(z)
}
```

```
foo()
```

4

Sub functions

Functions called within a function (ie subfunctions) must be defined within that function to access any variables defined in the local environment without being passed.

This fails:

```
bar <- function() {
    z <- x + y
    return(z)
}

foo <- function() {
    y <- 3
    z <- bar()
    return(z)
}

foo()
```

Error in bar() : object 'y' not found

This works:

```
foo <- function() {

    bar <- function() {
        z <- x + y
        return(z)
    }

    y <- 3
    z <- bar()
    return(z)
}

foo()
```

4

Global Assignment

Variables can be assigned globally from any environment using `<<-`. `bar()` can now access `y`.

```
bar <- function() {
    z <- x + y
    return(z)
}

foo <- function() {
```


www.dbooks.org

```
    y <<- 3
    z <- bar()
    return(z)
}

foo()
```

4

Global assignment is highly discouraged. Use of a wrapper function or explicitly calling variables from another local environment is greatly preferred.

Explicit Assignment of Environments and Variables

Environments in R can be explicitly call and named. Variables can be explicitly assigned and call to or from those environments.

A commonly created environment is one which encloses `package:base` or a subenvironment within `package:base`.

```
e1 <- new.env(parent = baseenv())
e2 <- new.env(parent = e1)
```

Variables can be explicitly assigned and call to or from those environments.

```
assign("a", 3, envir = e1)
   get("a", envir = e1)
   get("a", envir = e2)
```

3

3

Since `e2` inherits from `e1`, `a` is `3` in both `e1` and `e2`. However, assigning `a` within `e2` does not change the value of `a` in `e1`.

```
assign("a", 2, envir = e2)
   get("a", envir = e2)
   get("a", envir = e1)
```

3

2

Function Exit

The `on.exit()` function is handy for variable clean up if global variables must be assigned.

Some parameters, especially those for graphics, can only be set globally. This small function is common when creating more specialized plots.

```
new_plot <- function(...) {

     old_pars <- par(mar = c(5,4,4,2) + .1, mfrow = c(1,1))
     on.exit(par(old_pars))
     plot(...)
 }
```

Packages and Masking

Functions and objects in different packages may have the same name. The package loaded later will 'mask' the earlier package and a warning message will be printed. When calling the function by name, the function from the most recently loaded package will be run. The earlier function can be accessed explicitly.

```
library(plyr)
library(dplyr)
```

> Attaching package: ‘dplyr’
>
> The following objects are masked from ‘package:plyr’:
>
> arrange, count, desc, failwith, id, mutate, rename, summarise, summarize
>
> The following objects are masked from ‘package:stats’:
>
> filter, lag
>
> The following objects are masked from ‘package:base’:
>
> intersect, setdiff, setequal, union

When writing code, it is always best practice to call functions explicitly using `package::function()` specifically to avoid this issue.

Read Scope of variables online: https://riptutorial.com/r/topic/3138/scope-of-variables

www.dbooks.org

Chapter 104: Set operations

Remarks

A set contains only one copy of each distinct element. Unlike some other programming languages, base R does not have a dedicated data type for sets. Instead, R treats a vector like a set by taking only its distinct elements. This applies to the set operators, `setdiff`, `intersect`, `union`, `setequal` and `%in%`. For `v %in% S`, only `S` is treated as a set, however, not the vector `v`.

For a true set data type in R, the Rcpp package provides some options.

Examples

Set operators for pairs of vectors

Comparing sets

In R, a vector may contain duplicated elements:

```
v = "A"
w = c("A", "A")
```

However, a set contains only one copy of each element. R treats a vector like a set by taking only its distinct elements, so the two vectors above are regarded as the same:

```
setequal(v, w)
# TRUE
```

Combining sets

The key functions have natural names:

```
x = c(1, 2, 3)
y = c(2, 4)

union(x, y)
# 1 2 3 4

intersect(x, y)
# 2

setdiff(x, y)
# 1 3
```

These are all documented on the same page, `?union`.

Set membership for vectors

The `%in%` operator compares a vector with a set.

```
v = "A"
w = c("A", "A")

w %in% v
# TRUE TRUE

v %in% w
# TRUE
```

Each element on the left is treated individually and tested for membership in the set associated with the vector on the right (consisting of all its distinct elements).

Unlike equality tests, `%in%` always returns `TRUE` or `FALSE`:

```
c(1, NA) %in% c(1, 2, 3, 4)
# TRUE FALSE
```

The documentation is at `?`%in%``.

Cartesian or "cross" products of vectors

To find every vector of the form (x, y) where x is drawn from vector X and y from Y, we use `expand.grid`:

```
X = c(1, 1, 2)
Y = c(4, 5)

expand.grid(X, Y)

#   Var1 Var2
# 1    1    4
# 2    1    4
# 3    2    4
# 4    1    5
# 5    1    5
# 6    2    5
```

The result is a data.frame with one column for each vector passed to it. Often, we want to take the Cartesian product of sets rather than to expand a "grid" of vectors. We can use `unique`, `lapply` and `do.call`:

```
m = do.call(expand.grid, lapply(list(X, Y), unique))

#   Var1 Var2
# 1    1    4
# 2    2    4
# 3    1    5
# 4    2    5
```


www.dbooks.org

Applying functions to combinations

If you then want to apply a function to each resulting combination `f(x,y)`, it can be added as another column:

```
m$p = with(m, Var1*Var2)
#   Var1 Var2  p
# 1    1    4  4
# 2    2    4  8
# 3    1    5  5
# 4    2    5 10
```

This approach works for as many vectors as we need, but in the special case of two, it is sometimes a better fit to have the result in a matrix, which can be achieved with `outer`:

```
uX = unique(X)
uY = unique(Y)

outer(setNames(uX, uX), setNames(uY, uY), `*`)

#   4  5
# 1 4  5
# 2 8 10
```

For related concepts and tools, see the combinatorics topic.

Make unique / drop duplicates / select distinct elements from a vector

`unique` drops duplicates so that each element in the result is unique (only appears once):

```
x = c(2, 1, 1, 2, 1)

unique(x)
# 2 1
```

Values are returned in the order they first appeared.

`duplicated` tags each duplicated element:

```
duplicated(x)
# FALSE FALSE TRUE TRUE TRUE
```

`anyDuplicated(x) > 0L` is a quick way of checking whether a vector contains any duplicates.

Measuring set overlaps / Venn diagrams for vectors

To count how many elements of two sets overlap, one could write a custom function:

```
xtab_set <- function(A, B){
    both    <-  union(A, B)
```

```
    inA     <-  both %in% A
    inB     <-  both %in% B
    return(table(inA, inB))
}

A = 1:20
B = 10:30

xtab_set(A, B)

#        inB
# inA     FALSE TRUE
#   FALSE     0   10
#   TRUE      9   11
```

A Venn diagram, offered by various packages, can be used to visualize overlap counts across multiple sets.

Read Set operations online: https://riptutorial.com/r/topic/1383/set-operations

www.dbooks.org

Chapter 105: Shiny

Examples

Create an app

Shiny is an R package developed by RStudio that allows the creation of web pages to interactively display the results of an analysis in R.

There are two simple ways to create a Shiny app:

- in one `.R` file, or
- in two files: `ui.R` and `server.R`.

A Shiny app is divided into two parts:

- **ui**: A user interface script, controlling the layout and appearance of the application.
- **server**: A server script which contains code to allow the application to react.

One file

```
library(shiny)

# Create the UI
ui <- shinyUI(fluidPage(
    # Application title
    titlePanel("Hello World!")
))

# Create the server function
server <- shinyServer(function(input, output){})

# Run the app
shinyApp(ui = ui, server = server)
```

Two files

Create `ui.R` file

```
library(shiny)

# Define UI for application
shinyUI(fluidPage(
  # Application title
  titlePanel("Hello World!")
))
```

Create `server.R` file

```
library(shiny)

# Define server logic
shinyServer(function(input, output){})
```

Radio Button

You can create a set of radio buttons used to select an item from a list.

It's possible to change the settings :

- selected : The initially selected value (character(0) for no selection)
- inline : horizontal or vertical
- width

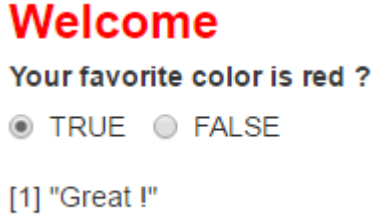
It is also possible to add HTML.

```
library(shiny)

ui <- fluidPage(
  radioButtons("radio",
               label = HTML('<FONT color="red"><FONT size="5pt">Welcome</FONT></FONT><br>
<b>Your favorite color is red ?</b>'),
               choices = list("TRUE" = 1, "FALSE" = 2),
               selected = 1,
               inline = T,
               width = "100%"),
  fluidRow(column(3, textOutput("value"))))


server <- function(input, output){
  output$value <- renderPrint({
    if(input$radio == 1){return('Great !')}
     else{return("Sorry !")}})}

shinyApp(ui = ui, server = server)
```



Checkbox Group

Create a group of checkboxes that can be used to toggle multiple choices independently. The server will receive the input as a character vector of the selected values.

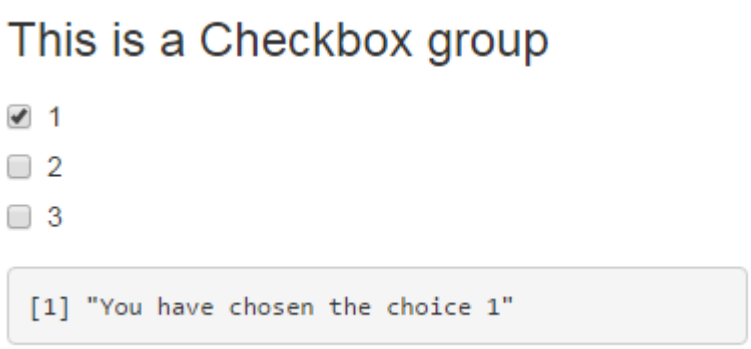
www.dbooks.org

```
library(shiny)

ui <- fluidPage(
  checkboxGroupInput("checkGroup1", label = h3("This is a Checkbox group"),
                 choices = list("1" = 1, "2" = 2, "3" = 3),
                 selected = 1),
  fluidRow(column(3, verbatimTextOutput("text_choice")))
  )


server <- function(input, output){
  output$text_choice <- renderPrint({
    return(paste0("You have chosen the choice ",input$checkGroup1))})
}

shinyApp(ui = ui, server = server)
```



It's possible to change the settings :

- label : title
- choices : selected values
- selected : The initially selected value (NULL for no selection)
- inline : horizontal or vertical
- width

It is also possible to add HTML.

Select box

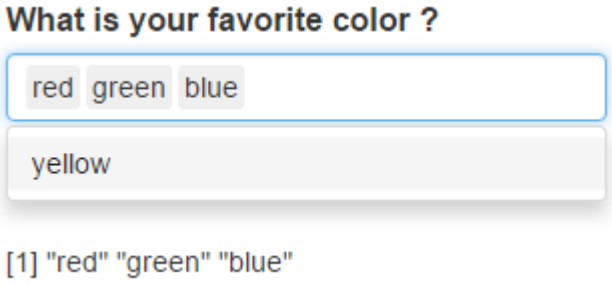
Create a select list that can be used to choose a single or multiple items from a list of values.

```
library(shiny)

ui <- fluidPage(
  selectInput("id_selectInput",
          label = HTML('<B><FONT size="3">What is your favorite color ?</FONT></B>'),
          multiple = TRUE,
          choices = list("red" = "red", "green" = "green", "blue" = "blue", "yellow" =
"yellow"),
          selected = NULL),
  br(), br(),
  fluidRow(column(3, textOutput("text_choice"))))
```

```
server <- function(input, output){
  output$text_choice <- renderPrint({
    return(input$id_selectInput)})
}

shinyApp(ui = ui, server = server)
```




```
[1] "red" "green" "blue"
```

It's possible to change the settings :

- label : title
- choices : selected values
- selected : The initially selected value (NULL for no selection)
- multiple : TRUE or FALSE
- width
- size
- selectize: TRUE or FALSE (for use or not selectize.js, change the display)

It is also possible to add HTML.

Launch a Shiny app

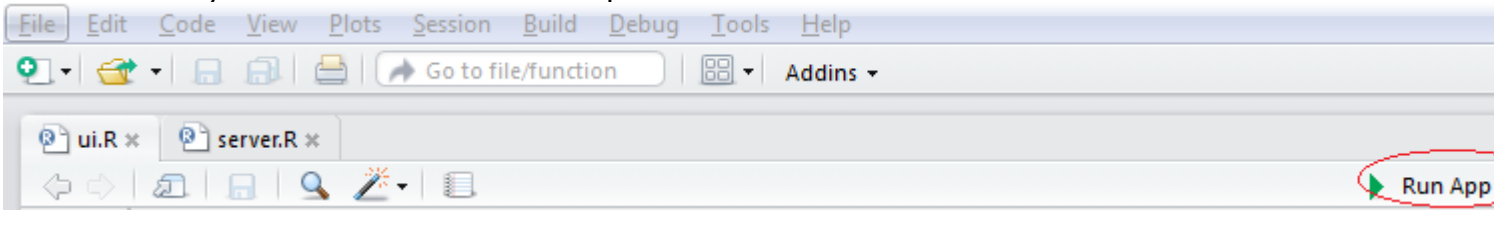
You can launch an application in several ways, depending on how you create you app. If your app is divided in two files `ui.R` and `server.R` or if all of your app is in one file.

1. Two files app

Your two files `ui.R` and `server.R`have to be in the same folder. You could then launch your app by running in the console the `shinyApp()` function and by passing the path of the directory that contains the Shiny app.

```
shinyApp("path_to_the_folder_containing_the_files")
```

You can also launch the app directly from Rstudio by pressing the **Run App** button that appear on Rstudio when you an `ui.R` or `server.R` file open.



www.dbooks.org

Or you can simply write `runApp()` on the console if your working directory is Shiny App directory.

2. One file app

If you create your in one `R` file you can also launch it with the `shinyApp()` function.

- inside of your code :

```
library(shiny)

ui <- fluidPage() #Create the ui
server <- function(input, output){} #create the server

shinyApp(ui = ui, server = server) #run the App
```

- in the console by adding path to a `.R` file containing the Shiny application with the paramter `appFile`:

```
shinyApp(appFile="path_to_my_R_file_containig_the_app")
```

Control widgets

Function	Widget
actionButton	Action Button
checkboxGroupInput	A group of check boxes
checkboxInput	A single check box
dateInput	A calendar to aid date selection
dateRangeInput	A pair of calendars for selecting a date range
fileInput	A file upload control wizard
helpText	Help text that can be added to an input form
numericInput	A field to enter numbers
radioButtons	A set of radio buttons
selectInput	A box with choices to select from
sliderInput	A slider bar
submitButton	A submit button
textInput	A field to enter text

```
library(shiny)

# Create the UI
ui <- shinyUI(fluidPage(
  titlePanel("Basic widgets"),

  fluidRow(

    column(3,
           h3("Buttons"),
           actionButton("action", label = "Action"),
           br(),
           br(),
           submitButton("Submit")),

    column(3,
           h3("Single checkbox"),
           checkboxInput("checkbox", label = "Choice A", value = TRUE)),

    column(3,
           checkboxGroupInput("checkGroup",
                              label = h3("Checkbox group"),
                              choices = list("Choice 1" = 1,
                                             "Choice 2" = 2, "Choice 3" = 3),
                              selected = 1)),

    column(3,
           dateInput("date",
                     label = h3("Date input"),
                     value = "2014-01-01"))
  ),

  fluidRow(

    column(3,
           dateRangeInput("dates", label = h3("Date range"))),

    column(3,
           fileInput("file", label = h3("File input"))),

    column(3,
           h3("Help text"),
           helpText("Note: help text isn't a true widget,",
                    "but it provides an easy way to add text to",
                    "accompany other widgets.")),

    column(3,
           numericInput("num",
                        label = h3("Numeric input"),
                        value = 1))
  ),

  fluidRow(

    column(3,
           radioButtons("radio", label = h3("Radio buttons"),
                        choices = list("Choice 1" = 1, "Choice 2" = 2,
                                       "Choice 3" = 3),selected = 1)),

    column(3,
           selectInput("select", label = h3("Select box"),
```


www.dbooks.org

```
                        choices = list("Choice 1" = 1, "Choice 2" = 2,
                                       "Choice 3" = 3), selected = 1)),

    column(3,
           sliderInput("slider1", label = h3("Sliders"),
                       min = 0, max = 100, value = 50),
           sliderInput("slider2", "",
                       min = 0, max = 100, value = c(25, 75))
    ),

    column(3,
           textInput("text", label = h3("Text input"),
                     value = "Enter text..."))
  )

))

# Create the server function
server <- shinyServer(function(input, output){})

# Run the app
shinyApp(ui = ui, server = server)
```

Debugging

`debug()` and `debugonce()` won't work well in the context of most Shiny debugging. However, `browser()` statements inserted in critical places can give you a lot of insight into how your Shiny code is (not) working. See also: Debugging using `browser()`

Showcase mode

Showcase mode displays your app alongside the code that generates it and highlights lines of code in server.R as it runs them.

There are two ways to enable Showcase mode:

- Launch Shiny app with the argument display.mode = "showcase", e.g., `runApp("MyApp", display.mode = "showcase")`.
- Create file called `DESCRIPTION` in your Shiny app folder and add this line in it: `DisplayMode: Showcase`.

Reactive Log Visualizer

Reactive Log Visualizer provides an interactive browser-based tool for visualizing reactive dependencies and execution in your application. To enable Reactive Log Visualizer, execute `options(shiny.reactlog=TRUE)` in R console and or add that line of code in your server.R file. To start Reactive Log Visualizer, hit Ctrl+F3 on Windows or Command+F3 on Mac when your app is running. Use left and right arrow keys to navigate in Reactive Log Visualizer.

Read Shiny online: https://riptutorial.com/r/topic/2044/shiny

Chapter 106: Solving ODEs in R

Syntax

- ode(y, times, func, parms, method, ...)

Parameters

Parameter	Details
y	(named) numeric vector: the initial (state) values for the ODE system
times	time sequence for which output is wanted; the first value of times must be the initial time
func	name of the function that computes the values of the derivatives in the ODE system
parms	(named) numeric vector: parameters passed to func
method	the integrator to use, by default: lsoda

Remarks

Note that it is necessary to return the rate of change in the same ordering as the specification of the state variables. In example "The Lorenz model" this means, that in the function "Lorenz" command

```
return(list(c(dX, dY, dZ)))
```

has the same order as the definition of the state variables

```
yini <- c(X = 1, Y = 1, Z = 1)
```

Examples

The Lorenz model

The Lorenz model describes the dynamics of three state variables, X, Y and Z. The model equations are:

$$\frac{dX}{dt} = a \cdot X + Y \cdot Z$$

www.dbooks.org

$$\frac{dY}{dt} = b \cdot (Y - Z)$$

$$\frac{dZ}{dT} = -X \cdot Y + c \cdot Y - Z$$

The initial conditions are:

$$X(0) = Y(0) = Z(0) = 1$$

and a, b and c are three parameters with

$$a = -8/3$$

$$b = -10$$

$$c = 28$$

```
library(deSolve)

## ------------------------------------------------------------------------
## Define R-function
## ------------------------------------------------------------------------

Lorenz <- function (t, y, parms) {
  with(as.list(c(y, parms)), {
    dX <- a * X + Y * Z
    dY <- b * (Y - Z)
    dZ <- -X * Y + c * Y - Z

    return(list(c(dX, dY, dZ)))
  })
}

## ------------------------------------------------------------------------
## Define parameters and variables
## ------------------------------------------------------------------------

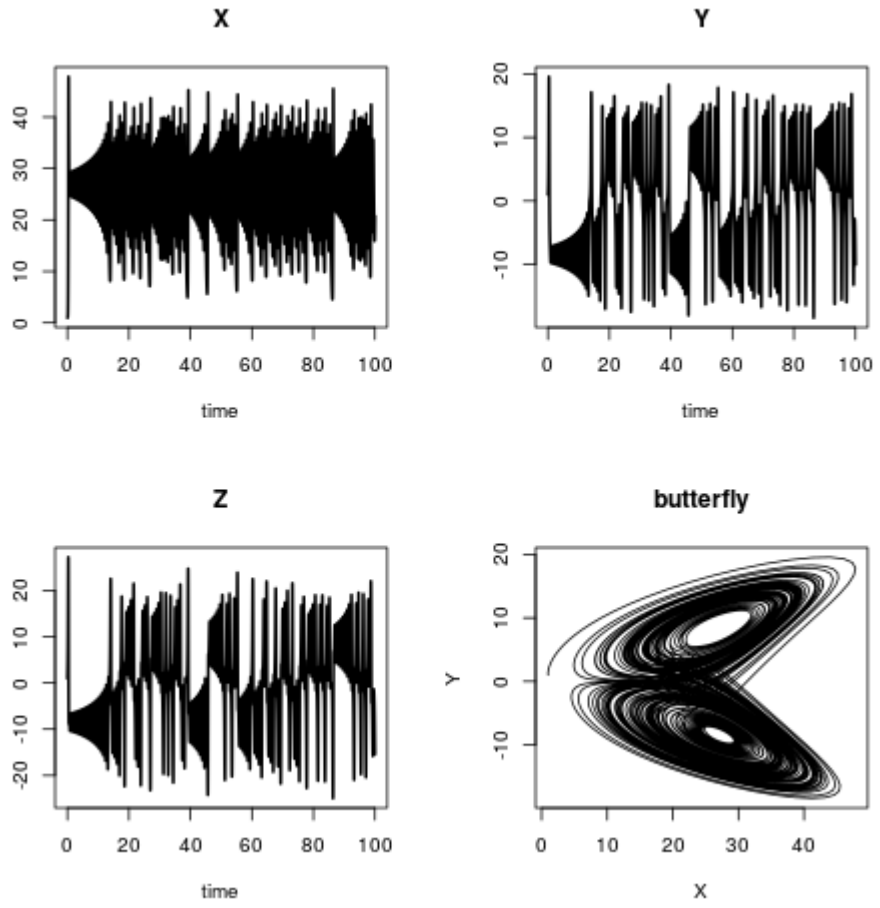
parms <- c(a = -8/3, b = -10, c = 28)
yini <- c(X = 1, Y = 1, Z = 1)
times <- seq(from = 0, to = 100, by = 0.01)


## ------------------------------------------------------------------------
## Solve the ODEs
## ------------------------------------------------------------------------

out <- ode(y = yini, times = times, func = Lorenz, parms = parms)

## ------------------------------------------------------------------------
## Plot the results
## ------------------------------------------------------------------------

plot(out, lwd = 2)
plot(out[,"X"], out[,"Y"],
     type = "l", xlab = "X",
     ylab = "Y", main = "butterfly")
```



Lotka-Volterra or: Prey vs. predator

```
library(deSolve)

## ------------------------------------------------------------------------
## Define R-function
## ------------------------------------------------------------------------

LV <- function(t, y, parms) {
    with(as.list(c(y, parms)), {

        dP <- rG * P * (1 - P/K) - rI * P * C
        dC <- rI * P * C * AE - rM * C

        return(list(c(dP, dC), sum = C+P))
    })
}

## ------------------------------------------------------------------------
## Define parameters and variables
## ------------------------------------------------------------------------

parms <- c(rI = 0.2, rG = 1.0, rM = 0.2, AE = 0.5, K = 10)
yini <- c(P = 1, C = 2)
times <- seq(from = 0, to = 200, by = 1)

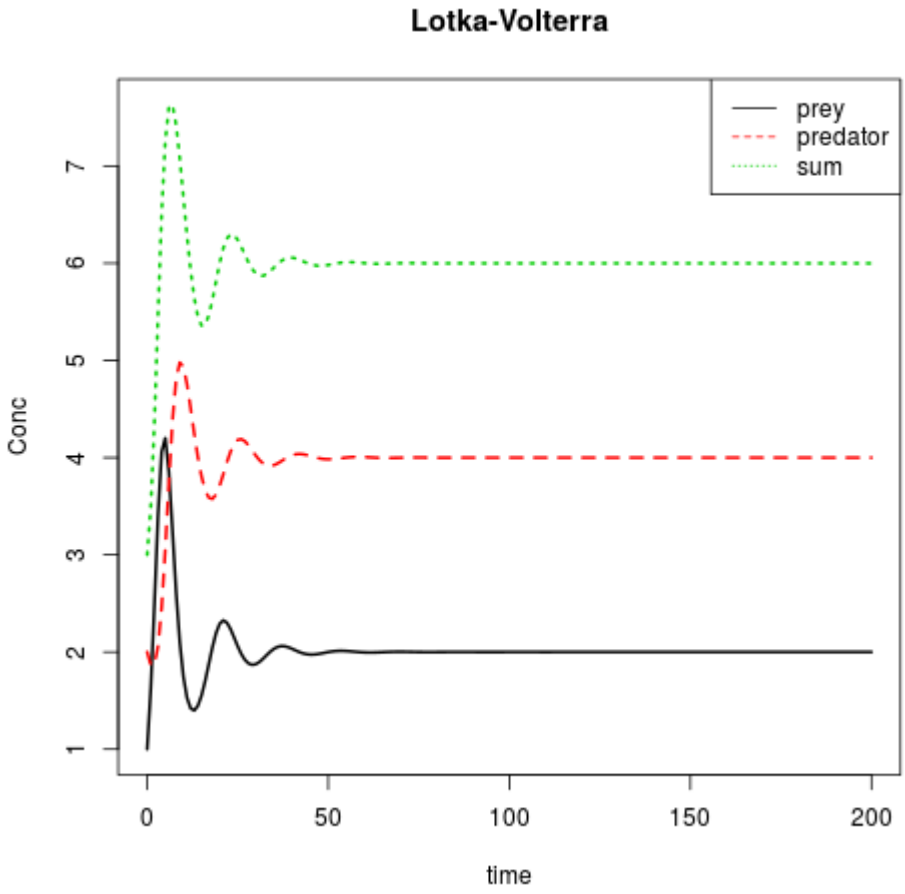
## ------------------------------------------------------------------------
## Solve the ODEs
## ------------------------------------------------------------------------
```


www.dbooks.org

```
out <- ode(y = yini, times = times, func = LV, parms = parms)

## ------------------------------------------------------------------------
## Plot the results
## ------------------------------------------------------------------------

matplot(out[ ,1], out[ ,2:4], type = "l", xlab = "time", ylab = "Conc",
        main = "Lotka-Volterra", lwd = 2)
legend("topright", c("prey", "predator", "sum"), col = 1:3, lty = 1:3)
```



ODEs in compiled languages - definition in R

```
library(deSolve)

## ------------------------------------------------------------------------
## Define parameters and variables
## ------------------------------------------------------------------------

eps <- 0.01;
M <- 10
k <- M * eps^2/2
L <- 1
L0 <- 0.5
r <- 0.1
w <- 10
g <- 1
```

```
parameter <- c(eps = eps, M = M, k = k, L = L, L0 = L0, r = r, w = w, g = g)

yini <- c(xl = 0, yl = L0, xr = L, yr = L0,
          ul = -L0/L, vl = 0,
          ur = -L0/L, vr = 0,
          lam1 = 0, lam2 = 0)


times <- seq(from = 0, to = 3, by = 0.01)

## ------------------------------------------------------------------------
## Define R-function
## ------------------------------------------------------------------------

caraxis_R <- function(t, y, parms) {
  with(as.list(c(y, parms)), {

    yb <- r * sin(w * t)
    xb <- sqrt(L * L - yb * yb)
    Ll <- sqrt(xl^2 + yl^2)
    Lr <- sqrt((xr - xb)^2 + (yr - yb)^2)

    dxl <- ul; dyl <- vl; dxr <- ur; dyr <- vr

    dul  <- (L0-Ll) * xl/Ll      + 2 * lam2 * (xl-xr) + lam1*xb
    dvl  <- (L0-Ll) * yl/Ll      + 2 * lam2 * (yl-yr) + lam1*yb - k * g

    dur  <- (L0-Lr) * (xr-xb)/Lr - 2 * lam2 * (xl-xr)
    dvr  <- (L0-Lr) * (yr-yb)/Lr - 2 * lam2 * (yl-yr) - k * g

    c1   <- xb * xl + yb * yl
    c2   <- (xl - xr)^2 + (yl - yr)^2 - L * L

    return(list(c(dxl, dyl, dxr, dyr, dul, dvl, dur, dvr, c1, c2)))
  })
}
```

ODEs in compiled languages - definition in C

```
sink("caraxis_C.c")
cat("
/* suitable names for parameters and state variables */

#include <R.h>
#include <math.h>
static double parms[8];

#define eps parms[0]
#define m   parms[1]
#define k   parms[2]
#define L   parms[3]
#define L0  parms[4]
#define r   parms[5]
#define w   parms[6]
#define g   parms[7]

/*-------------------------------------------------------------------
 initialising the parameter common block
-------------------------------------------------------------------
```


www.dbooks.org

```
*/
void init_C(void (* daeparms)(int *, double *)) {
  int N = 8;
  daeparms(&N, parms);
    }
/* Compartments */

#define xl y[0]
#define yl y[1]
#define xr y[2]
#define yr y[3]
#define lam1 y[8]
#define lam2 y[9]

/*-----------------------------------------------------------------------
 the residual function
-----------------------------------------------------------------------
*/
void caraxis_C (int *neq, double *t, double *y, double *ydot,
              double *yout, int* ip)
{
  double yb, xb, Lr, Ll;

  yb  = r * sin(w * *t) ;
  xb  = sqrt(L * L - yb * yb);
  Ll  = sqrt(xl * xl + yl * yl) ;
  Lr  = sqrt((xr-xb)*(xr-xb) + (yr-yb)*(yr-yb));

  ydot[0] = y[4];
  ydot[1] = y[5];
  ydot[2] = y[6];
  ydot[3] = y[7];

  ydot[4] = (L0-Ll) * xl/Ll + lam1*xb + 2*lam2*(xl-xr)    ;
  ydot[5] = (L0-Ll) * yl/Ll + lam1*yb + 2*lam2*(yl-yr) - k*g;
  ydot[6] = (L0-Lr) * (xr-xb)/Lr      - 2*lam2*(xl-xr)       ;
  ydot[7] = (L0-Lr) * (yr-yb)/Lr      - 2*lam2*(yl-yr) - k*g   ;

  ydot[8]  = xb * xl + yb * yl;
  ydot[9]  = (xl-xr) * (xl-xr) + (yl-yr) * (yl-yr) - L*L;

}
", fill = TRUE)
sink()
system("R CMD SHLIB caraxis_C.c")
dyn.load(paste("caraxis_C", .Platform$dynlib.ext, sep = ""))
dllname_C <- dyn.load(paste("caraxis_C", .Platform$dynlib.ext, sep = ""))[[1]]
```

ODEs in compiled languages - definition in fortran

```
sink("caraxis_fortran.f")
cat("
c-----------------------------------------------------------------
c Initialiser for parameter common block
c-----------------------------------------------------------------
       subroutine init_fortran(daeparms)

        external daeparms
        integer, parameter :: N = 8
```

```
        double precision parms(N)
        common /myparms/parms

        call daeparms(N, parms)
        return
        end

c-----------------------------------------------------------------------
c rate of change
c-----------------------------------------------------------------------
        subroutine caraxis_fortran(neq, t, y, ydot, out, ip)
        implicit none
        integer          neq, IP(*)
        double precision t, y(neq), ydot(neq), out(*)
        double precision eps, M, k, L, L0, r, w, g
        common /myparms/ eps, M, k, L, L0, r, w, g

        double precision xl, yl, xr, yr, ul, vl, ur, vr, lam1, lam2
        double precision yb, xb, Ll, Lr, dxl, dyl, dxr, dyr
        double precision dul, dvl, dur, dvr, c1, c2

c expand state variables
         xl = y(1)
         yl = y(2)
         xr = y(3)
         yr = y(4)
         ul = y(5)
         vl = y(6)
         ur = y(7)
         vr = y(8)
         lam1 = y(9)
         lam2 = y(10)

         yb = r * sin(w * t)
         xb = sqrt(L * L - yb * yb)
         Ll = sqrt(xl**2 + yl**2)
         Lr = sqrt((xr - xb)**2 + (yr - yb)**2)

         dxl = ul
         dyl = vl
         dxr = ur
         dyr = vr

         dul = (L0-Ll) * xl/Ll      + 2 * lam2 * (xl-xr) + lam1*xb
         dvl = (L0-Ll) * yl/Ll      + 2 * lam2 * (yl-yr) + lam1*yb - k*g
         dur = (L0-Lr) * (xr-xb)/Lr - 2 * lam2 * (xl-xr)
         dvr = (L0-Lr) * (yr-yb)/Lr - 2 * lam2 * (yl-yr) - k*g

         c1  = xb * xl + yb * yl
         c2  = (xl - xr)**2 + (yl - yr)**2 - L * L

c function values in ydot
         ydot(1)  = dxl
         ydot(2)  = dyl
         ydot(3)  = dxr
         ydot(4)  = dyr
         ydot(5)  = dul
         ydot(6)  = dvl
         ydot(7)  = dur
         ydot(8)  = dvr
         ydot(9)  = c1
```


www.dbooks.org

```
         ydot(10) = c2
        return
        end
", fill = TRUE)

sink()
system("R CMD SHLIB caraxis_fortran.f")
dyn.load(paste("caraxis_fortran", .Platform$dynlib.ext, sep = ""))
dllname_fortran <- dyn.load(paste("caraxis_fortran", .Platform$dynlib.ext, sep = ""))[[1]]
```

ODEs in compiled languages - a benchmark test

When you compiled and loaded the code in the three examples before (ODEs in compiled languages - definition in R, ODEs in compiled languages - definition in C and ODEs in compiled languages - definition in fortran) you are able to run a benchmark test.

```
library(microbenchmark)

R <- function(){
  out <- ode(y = yini, times = times, func = caraxis_R,
             parms = parameter)
}


C <- function(){
  out <- ode(y = yini, times = times, func = "caraxis_C",
             initfunc = "init_C", parms = parameter,
             dllname = dllname_C)
}

fortran <- function(){
  out <- ode(y = yini, times = times, func = "caraxis_fortran",
             initfunc = "init_fortran", parms = parameter,
             dllname = dllname_fortran)
}
```

Check if results are equal:

```
all.equal(tail(R()), tail(fortran()))
all.equal(R()[,2], fortran()[,2])
all.equal(R()[,2], C()[,2])
```

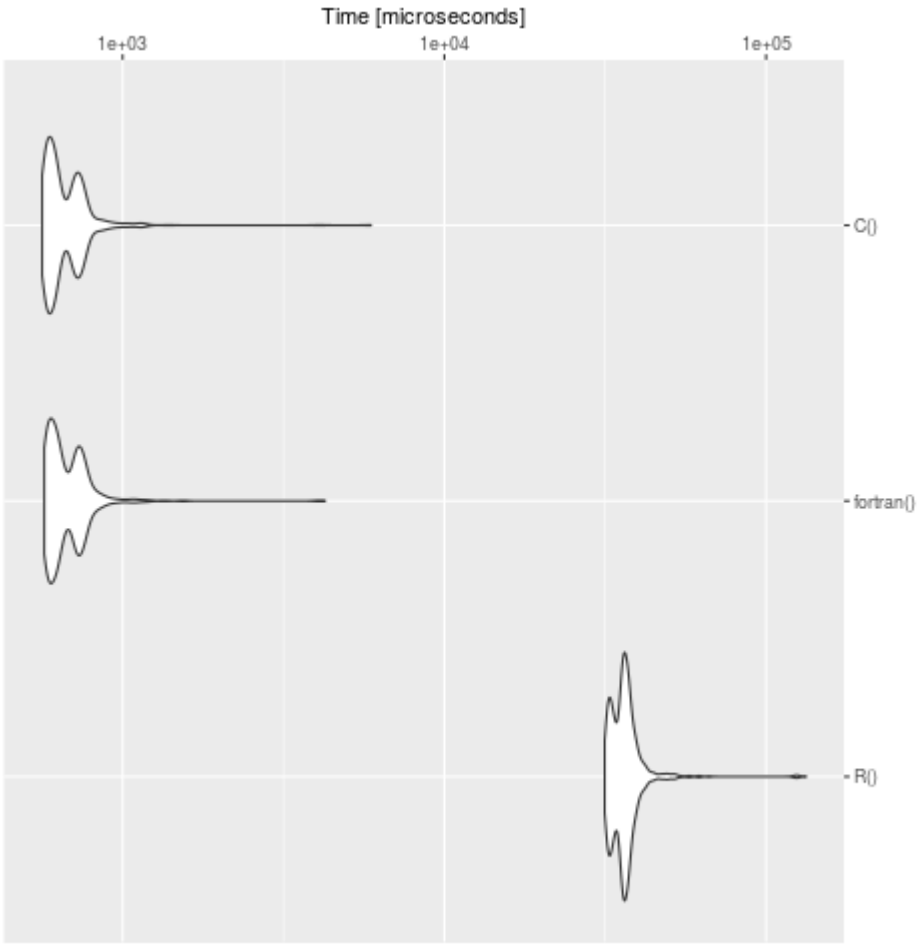
Make a benchmark (Note: On your machine the times are, of course, different):

```
bench <- microbenchmark::microbenchmark(
  R(),
  fortran(),
  C(),
  times = 1000
)

summary(bench)

     expr         min        lq       mean     median         uq        max neval cld
      R()   31508.928 33651.541 36747.8733 36062.2475 37546.8025 132996.564  1000   b
fortran()     570.674   596.700   686.1084   637.4605   730.1775   4256.555  1000  a
```

```
C()      562.163   590.377   673.6124   625.0700   723.8460   5914.347  1000  a
```



We see clearly, that R is slow in contrast to the definition in C and fortran. For big models it's worth to translate the problem in a compiled language. The package `cOde` is one possibility to translate ODEs from R to C.

Read Solving ODEs in R online: https://riptutorial.com/r/topic/7448/solving-odes-in-r

www.dbooks.org

Chapter 107: Spark API (SparkR)

Remarks

The `SparkR` package let's you work with distributed data frames on top of a Spark cluster. These allow you to do operations like selection, filtering, aggregation on very large datasets. SparkR overview SparkR package documentation

Examples

Setup Spark context

Setup Spark context in R

To start working with Sparks distributed dataframes, you must connect your R program with an existing Spark Cluster.

```
library(SparkR)
sc <- sparkR.init() # connection to Spark context
sqlContext <- sparkRSQL.init(sc) # connection to SQL context
```

Here are infos how to connect your IDE to a Spark cluster.

Get Spark Cluster

There is an Apache Spark introduction topic with install instructions. Basically, you can employ a Spark Cluster locally via java (see instructions) or use (non-free) cloud applications (e.g. Microsoft Azure [topic site], IBM).

Cache data

What:

Caching can optimize computation in Spark. Caching stores data in memory and is a special case of persistence. Here is explained what happens when you cache an RDD in Spark.

Why:

Basically, caching saves an interim partial result - usually after transformations - of your original data. So, when you use the cached RDD, the already transformed data from memory is accessed without recomputing the earlier transformations.

How:

Here is an example how to quickly access large data *(here 3 GB big csv)* from in-memory storage

when accessing it more then once:

```
library(SparkR)
# next line is needed for direct csv import:
Sys.setenv('SPARKR_SUBMIT_ARGS'='"--packages" "com.databricks:spark-csv_2.10:1.4.0" "sparkr-
shell"')
sc <- sparkR.init()
sqlContext <- sparkRSQL.init(sc)

# loading 3 GB big csv file:
train <- read.df(sqlContext, "/train.csv", source = "com.databricks.spark.csv", inferSchema =
"true")
cache(train)
system.time(head(train))
# output: time elapsed: 125 s. This action invokes the caching at this point.
system.time(head(train))
# output: time elapsed: 0.2 s (!!)
```

Create RDDs (Resilient Distributed Datasets)

From dataframe:

```
mtrdd <- createDataFrame(sqlContext, mtcars)
```

From csv:

For csv's, you need to add the csv package to the environment before initiating the Spark context:

```
Sys.setenv('SPARKR_SUBMIT_ARGS'='"--packages" "com.databricks:spark-csv_2.10:1.4.0" "sparkr-
shell"') # context for csv import read csv ->
sc <- sparkR.init()
sqlContext <- sparkRSQL.init(sc)
```

Then, you can load the csv either by infering the data schema of the data in the columns:

```
train <- read.df(sqlContext, "/train.csv", header= "true", source =
"com.databricks.spark.csv", inferSchema = "true")
```

Or by specifying the data schema beforehand:

```
 customSchema <- structType(
    structField("margin", "integer"),
    structField("gross", "integer"),
    structField("name", "string"))

 train <- read.df(sqlContext, "/train.csv", header= "true", source =
"com.databricks.spark.csv", schema = customSchema)
```

Read Spark API (SparkR) online: https://riptutorial.com/r/topic/5349/spark-api--sparkr-

www.dbooks.org

Chapter 108: spatial analysis

Examples

Create spatial points from XY data set

When it comes to geographic data, R shows to be a powerful tool for data handling, analysis and visualisation.

Often, spatial data is avaliable as an XY coordinate data set in tabular form. This example will show how to create a spatial data set from an XY data set.

The packages `rgdal` and `sp` provide powerful functions. Spatial data in R can be stored as `Spatial*DataFrame` (where `*` can be `Points`, `Lines` or `Polygons`).

This example uses data which can be downloaded at OpenGeocode.

At first, the working directory has to be set to the folder of the downloaded CSV data set. Furthermore, the package `rgdal` has to be loaded.

```
setwd("D:/GeocodeExample/")
library(rgdal)
```

Afterwards, the CSV file storing cities and their geographical coordinates is loaded into R as a `data.frame`

```
xy <- read.csv("worldcities.csv", stringsAsFactors = FALSE)
```

Often, it is useful to get a glimpse of the data and its structure (e.g. column names, data types etc.).

```
head(xy)
str(xy)
```

This shows that the latitude and longitude columns are interpreted as character values, since they hold entries like "-33.532". Yet, the later used function `SpatialPointsDataFrame()` which creates the spatial data set requires the coordinate values to be of the data type `numeric`. Thus the two columns have to be converted.

```
xy$latitude <- as.numeric(xy$latitude)
xy$longitude <- as.numeric(xy$longitude)
```

Few of the values cannot be converted into numeric data and thus, `NA` values are created. They have to be removed.

```
xy <- xy[!is.na(xy$longitude),]
```

Finally, the XY data set can be converted into a spatial data set. This requires the coordinates and the specification of the Coordinate Refrence System (CRS) in which the coordinates are stored.

```
xySPoints <- SpatialPointsDataFrame(coords = c(xy[,c("longitude", "latitude")]),
proj4string = CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"),
data = xy
)
```

The basic plot function can easily be used to sneak peak the produced spatial points.

```
plot(xySPoints, pch = ".")
```



Importing a shape file (.shp)

rgdal

ESRI shape files can easily be imported into R by using the function `readOGR()` from the `rgdal` package.

```
library(rgdal)
shp <- readOGR(dsn = "/path/to/your/file", layer = "filename")
```

It is important to know, that the `dsn` must not end with `/` and the `layer` does not allow the file ending (e.g. `.shp`)

raster

www.dbooks.org

Another possible way of importing shapefiles is via the `raster` library and the `shapefile` function:

```
library(raster)
shp <- shapefile("path/to/your/file.shp")
```

Note how the path definition is different from the rgdal import statement.

tmap

`tmap` package provides a nice wrapper for the `rgdal::readORG` function.

```
library(tmap)
sph <- read_shape("path/to/your/file.shp")
```

Read spatial analysis online: https://riptutorial.com/r/topic/2093/spatial-analysis

Chapter 109: Speeding up tough-to-vectorize code

Examples

Speeding tough-to-vectorize for loops with Rcpp

Consider the following tough-to-vectorize for loop, which creates a vector of length `len` where the first element is specified (`first`) and each element `x_i` is equal to `cos(x_{i-1} + 1)`:

```
repeatedCosPlusOne <- function(first, len) {
  x <- numeric(len)
  x[1] <- first
  for (i in 2:len) {
    x[i] <- cos(x[i-1] + 1)
  }
  return(x)
}
```

This code involves a for loop with a fast operation (`cos(x[i-1]+1)`), which often benefit from vectorization. However, it is not trivial to vectorize this operation with base R, since R does not have a "cumulative cosine of x+1" function.

One possible approach to speeding this function would be to implement it in C++, using the Rcpp package:

```
library(Rcpp)
cppFunction("NumericVector repeatedCosPlusOneRcpp(double first, int len) {
  NumericVector x(len);
  x[0] = first;
  for (int i=1; i < len; ++i) {
    x[i] = cos(x[i-1]+1);
  }
  return x;
}")
```

This often provides significant speedups for large computations while yielding the exact same results:

```
all.equal(repeatedCosPlusOne(1, 1e6), repeatedCosPlusOneRcpp(1, 1e6))
# [1] TRUE
system.time(repeatedCosPlusOne(1, 1e6))
#    user  system elapsed
#   1.274   0.015   1.310
system.time(repeatedCosPlusOneRcpp(1, 1e6))
#    user  system elapsed
#   0.028   0.001   0.030
```

In this case, the Rcpp code generates a vector of length 1 million in 0.03 seconds instead of 1.31

www.dbooks.org

seconds with the base R approach.

Speeding tough-to-vectorize for loops by byte compiling

Following the Rcpp example in this documentation entry, consider the following tough-to-vectorize function, which creates a vector of length `len` where the first element is specified (`first`) and each element `x_i` is equal to `cos(x_{i-1} + 1)`:

```
repeatedCosPlusOne <- function(first, len) {
  x <- numeric(len)
  x[1] <- first
  for (i in 2:len) {
    x[i] <- cos(x[i-1] + 1)
  }
  return(x)
}
```

One simple approach to speeding up such a function without rewriting a single line of code is byte compiling the code using the R compile package:

```
library(compiler)
repeatedCosPlusOneCompiled <- cmpfun(repeatedCosPlusOne)
```

The resulting function will often be significantly faster while still returning the same results:

```
all.equal(repeatedCosPlusOne(1, 1e6), repeatedCosPlusOneCompiled(1, 1e6))
# [1] TRUE
system.time(repeatedCosPlusOne(1, 1e6))
#    user  system elapsed
#   1.175   0.014   1.201
system.time(repeatedCosPlusOneCompiled(1, 1e6))
#    user  system elapsed
#   0.339   0.002   0.341
```

In this case, byte compiling sped up the tough-to-vectorize operation on a vector of length 1 million from 1.20 seconds to 0.34 seconds.

Remark

The essence of `repeatedCosPlusOne`, as the cumulative application of a single function, can be expressed more transparently with `Reduce`:

```
iterFunc <- function(init, n, func) {
  funcs <- replicate(n, func)
  Reduce(function(., f) f(.), funcs, init = init, accumulate = TRUE)
}
repeatedCosPlusOne_vec <- function(first, len) {
  iterFunc(first, len - 1, function(.) cos(. + 1))
}
```

`repeatedCosPlusOne_vec` may be regarded as a "vectorization" of `repeatedCosPlusOne`. However, it can be expected to be *slower* by a factor of 2:

```
library(microbenchmark)
microbenchmark(
  repeatedCosPlusOne(1, 1e4),
  repeatedCosPlusOne_vec(1, 1e4)
)
#> Unit: milliseconds
#>                              expr       min        lq     mean   median       uq      max
neval cld
#>      repeatedCosPlusOne(1, 10000)  8.349261  9.216724 10.22715 10.23095 11.10817 14.33763
100  a
#>  repeatedCosPlusOne_vec(1, 10000) 14.406291 16.236153 17.55571 17.22295 18.59085 24.37059
100   b
```

Read Speeding up tough-to-vectorize code online: https://riptutorial.com/r/topic/1203/speeding-up-tough-to-vectorize-code

www.dbooks.org

Chapter 110: Split function

Examples

Basic usage of split

`split` allows to divide a vector or a data.frame into buckets with regards to a factor/group variables. This ventilation into buckets takes the form of a list, that can then be used to apply group-wise computation (`for` loops or `lapply`/`sapply`).

First example shows the usage of `split` on a vector:

Consider following vector of letters:

```
testdata <- c("e", "o", "r", "g", "a", "y", "w", "q", "i", "s", "b", "v", "x", "h", "u")
```

Objective is to separate those letters into `voyels` and `consonants`, ie split it accordingly to letter type.

Let's first create a grouping vector:

```
 vowels <- c('a','e','i','o','u','y')
 letter_type <- ifelse(testdata %in% vowels, "vowels", "consonants")
```

Note that `letter_type` has the same length that our vector `testdata`. Now we can `split` this test data in the two groups, `vowels` and `consonants` :

```
split(testdata, letter_type)
#$consonants
#[1] "r" "g" "w" "q" "s" "b" "v" "x" "h"

#$vowels
#[1] "e" "o" "a" "y" "i" "u"
```

Hence, the result is a list which names are coming from our grouping vector/factor `letter_type`.

`split` has also a method to deal with data.frames.

Consider for instance `iris` data:

```
data(iris)
```

By using `split`, one can create a list containing one data.frame per iris specie (variable: Species):

```
> liris <- split(iris, iris$Species)
> names(liris)
[1] "setosa"     "versicolor" "virginica"
> head(liris$setosa)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

```
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

(contains only data for setosa group).

One example operation would be to compute correlation matrix per iris specie; one would then use `lapply`:

```
> (lcor <- lapply(liris, FUN=function(df) cor(df[,1:4])))

    $setosa
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.7425467    0.2671758   0.2780984
Sepal.Width     0.7425467   1.0000000    0.1777000   0.2327520
Petal.Length    0.2671758   0.1777000    1.0000000   0.3316300
Petal.Width     0.2780984   0.2327520    0.3316300   1.0000000

$versicolor
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.5259107    0.7540490   0.5464611
Sepal.Width     0.5259107   1.0000000    0.5605221   0.6639987
Petal.Length    0.7540490   0.5605221    1.0000000   0.7866681
Petal.Width     0.5464611   0.6639987    0.7866681   1.0000000

$virginica
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.4572278    0.8642247   0.2811077
Sepal.Width     0.4572278   1.0000000    0.4010446   0.5377280
Petal.Length    0.8642247   0.4010446    1.0000000   0.3221082
Petal.Width     0.2811077   0.5377280    0.3221082   1.0000000
```

Then we can retrieve per group the best pair of correlated variables: (correlation matrix is reshaped/melted, diagonal is filtered out and selecting best record is performed)

```
> library(reshape)
> (topcor <- lapply(lcor, FUN=function(cormat){
   correlations <- melt(cormat,variable_name="correlatio);
   filtered <- correlations[correlations$X1 != correlations$X2,];
   filtered[which.max(filtered$correlation),]
}))

$setosa
           X1           X2     correlation
2 Sepal.Width Sepal.Length       0.7425467

$versicolor
            X1           X2     correlation
12 Petal.Width Petal.Length       0.7866681

$virginica
            X1           X2     correlation
3 Petal.Length Sepal.Length       0.8642247
```


www.dbooks.org

Note that one computations are performed on such groupwise level, one may be interested in stacking the results, which can be done with:

```
> (result <- do.call("rbind", topcor))

                     X1           X2     correlation
setosa      Sepal.Width Sepal.Length       0.7425467
versicolor  Petal.Width Petal.Length       0.7866681
virginica  Petal.Length Sepal.Length       0.8642247
```

Using split in the split-apply-combine paradigm

A popular form of data analysis is split-apply-combine, in which you split your data into groups, apply some sort of processing on each group, and then combine the results.

Let's consider a data analysis where we want to obtain the two cars with the best miles per gallon (mpg) for each cylinder count (cyl) in the built-in mtcars dataset. First, we split the `mtcars` data frame by the cylinder count:

```
(spl <- split(mtcars, mtcars$cyl))
# $`4`
#                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
# Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
# Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
# Merc 230       22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
# Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
# ...
#
# $`6`
#                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
# Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
# Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
# Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
# Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
# ...
#
# $`8`
#                     mpg cyl  disp  hp drat    wt  qsec vs am gear carb
# Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
# Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
# Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
# Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
# ...
```

This has returned a list of data frames, one for each cylinder count. As indicated by the output, we could obtain the relevant data frames with `spl$`4``, `spl$`6``, and `spl$`8`` (some might find it more visually appealing to use `spl$"4"` or `spl[["4"]]` instead).

Now, we can use `lapply` to loop through this list, applying our function that extracts the cars with the best 2 mpg values from each of the list elements:

```
(best2 <- lapply(spl, function(x) tail(x[order(x$mpg),], 2)))
# $`4`
#                 mpg cyl disp hp drat    wt  qsec vs am gear carb
```

```
# Fiat 128       32.4   4 78.7 66 4.08 2.200 19.47  1  1    4    1
# Toyota Corolla 33.9   4 71.1 65 4.22 1.835 19.90  1  1    4    1
#
# $`6`
#                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
# Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
# Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#
# $`8`
#                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
# Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
# Pontiac Firebird  19.2   8  400 175 3.08 3.845 17.05  0  0    3    2
```

Finally, we can combine everything together using `rbind`. We want to call `rbind(best2[["4"]], best2[["6"]], best2[["8"]])`, but this would be tedious if we had a huge list. As a result, we use:

```
do.call(rbind, best2)
#                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
# 4.Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
# 4.Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
# 6.Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
# 6.Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
# 8.Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
# 8.Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
```

This returns the result of `rbind` (argument 1, a function) with all the elements of `best2` (argument 2, a list) passed as arguments.

With simple analyses like this one, it can be more compact (and possibly much less readable!) to do the whole split-apply-combine in a single line of code:

```
do.call(rbind, lapply(split(mtcars, mtcars$cyl), function(x) tail(x[order(x$mpg),], 2)))
```

It is also worth noting that the `lapply(split(x,f), FUN)` combination can be alternatively framed using the `?by` function:

```
by(mtcars, mtcars$cyl, function(x) tail(x[order(x$mpg),], 2))
do.call(rbind, by(mtcars, mtcars$cyl, function(x) tail(x[order(x$mpg),], 2)))
```

Read Split function online: https://riptutorial.com/r/topic/1073/split-function

www.dbooks.org

Chapter 111: sqldf

Examples

Basic Usage Examples

`sqldf()` from the package `sqldf` allows the use of SQLite queries to select and manipulate data in R. SQL queries are entered as character strings.

To select the first 10 rows of the "diamonds" dataset from the package `ggplot2`, for example:

```
data("diamonds")
head(diamonds)
```

```
# A tibble: 6 x 10
  carat       cut color clarity depth table price     x     y     z
  <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23     Ideal     E     SI2  61.5    55   326  3.95  3.98  2.43
2  0.21   Premium     E     SI1  59.8    61   326  3.89  3.84  2.31
3  0.23      Good     E     VS1  56.9    65   327  4.05  4.07  2.31
4  0.29   Premium     I     VS2  62.4    58   334  4.20  4.23  2.63
5  0.31      Good     J     SI2  63.3    58   335  4.34  4.35  2.75
6  0.24 Very Good     J    VVS2  62.8    57   336  3.94  3.96  2.48
```

```
require(sqldf)
sqldf("select * from diamonds limit 10")
```

```
   carat       cut color clarity depth table price    x    y    z
1   0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
2   0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
3   0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
4   0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
5   0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
6   0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
7   0.24 Very Good     I    VVS1  62.3    57   336 3.95 3.98 2.47
8   0.26 Very Good     H     SI1  61.9    55   337 4.07 4.11 2.53
9   0.22      Fair     E     VS2  65.1    61   337 3.87 3.78 2.49
10  0.23 Very Good     H     VS1  59.4    61   338 4.00 4.05 2.39
```

To select the first 10 rows where for the color "E":

```
sqldf("select * from diamonds where color = 'E' limit 10")
```

```
   carat       cut color clarity depth table price    x    y    z
1   0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
2   0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
3   0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
4   0.22      Fair     E     VS2  65.1    61   337 3.87 3.78 2.49
5   0.20   Premium     E     SI2  60.2    62   345 3.79 3.75 2.27
6   0.32   Premium     E      I1  60.9    58   345 4.38 4.42 2.68
```

```
7   0.23 Very Good     E     VS2  63.8    55   352 3.85 3.92 2.48
8   0.23 Very Good     E     VS1  60.7    59   402 3.97 4.01 2.42
9   0.23 Very Good     E     VS1  59.5    58   402 4.01 4.06 2.40
10  0.23      Good     E     VS1  64.1    59   402 3.83 3.85 2.46
```

Notice in the example above that quoted strings within the SQL query are quoted using '' if the overall query is quoted with "" (this also works in reverse).

Suppose that we wish to add a new column to count the number of Premium cut diamonds over 1 carat:

```
sqldf("select count(*) from diamonds where carat > 1 and color = 'E'")
```

```
  count(*)
1     1892
```

Results of created values can also be returned as new columns:

```
sqldf("select *, count(*) as cnt_big_E_colored_stones from diamonds where carat > 1 and color
= 'E' group by clarity")
```

```
  carat       cut color clarity depth table price    x    y    z
cnt_big_E_colored_stones
1  1.30      Fair     E      I1  66.5    58  2571 6.79 6.75 4.50
65
2  1.28     Ideal     E      IF  60.7    57 18700 7.09 6.99 4.27
28
3  2.02 Very Good     E     SI1  59.8    59 18731 8.11 8.20 4.88
499
4  2.03   Premium     E     SI2  61.5    59 18477 8.24 8.16 5.04
666
5  1.51     Ideal     E     VS1  61.5    57 18729 7.34 7.40 4.53
158
6  1.72 Very Good     E     VS2  63.4    56 18557 7.65 7.55 4.82
318
7  1.20     Ideal     E    VVS1  61.8    56 16256 6.78 6.87 4.22
52
8  1.55     Ideal     E    VVS2  62.5    55 18188 7.38 7.40 4.62
106
```

If one would be interested what is the **max** `price` of the diamond **according** to the `cut`:

```
sqldf("select cut,  max(price) from diamonds group by cut")

        cut max(price)
1      Fair      18574
2      Good      18788
3     Ideal      18806
4   Premium      18823
5 Very Good      18818
```

Read sqldf online: https://riptutorial.com/r/topic/2100/sqldf

www.dbooks.org

Chapter 112: Standardize analyses by writing standalone R scripts

Introduction

If you want to routinely apply an R analysis to a lot of separate data files, or provide a repeatable analysis method to other people, an executable R script is a user-friendly way to do so. Instead of you or your user having to call `R` and execute your script inside R via `source(.)` or a function call, your user may simply call the script itself as if it was a program.

Remarks

To represent the standard input-/output channels, use the functions `file("stdin")` (input from terminal or other program via pipe), `stdout()` (standard output) and `stderr()` (standard error). Note that while there is the function `stdin()`, it can not be used when supplying a ready-made script to R, because it will read the next lines of that script instead of user input.

Examples

The basic structure of standalone R program and how to call it

The first standalone R script

Standalone R scripts are not executed by the program `R` (`R.exe` under Windows), but by a program called `Rscript` (`Rscript.exe`), which is included in your `R` installation by default.

To hint at this fact, standalone R scripts start with a special line called **Shebang line**, which holds the following content: `#!/usr/bin/env Rscript`. Under Windows, an additional measure is needed, which is detailled later.

The following simple standalone R script saves a histogram under the file name "hist.png" from numbers it receives as input:

```
#!/usr/bin/env Rscript

# User message (\n = end the line)
cat("Input numbers, separated by space:\n")
# Read user input as one string (n=1 -> Read only one line)
input <- readLines(file('stdin'), n=1)
# Split the string at each space (\\s == any space)
input <- strsplit(input, "\\s")[[1]]
# convert the obtained vector of strings to numbers
input <- as.numeric(input)

# Open the output picture file
```

```
png("hist.png",width=400, height=300)
# Draw the histogram
hist(input)
# Close the output file
dev.off()
```

You can see several key elements of a standalone R script. In the first line, you see the Shebang line. Followed by that, `cat("...\n")` is used to print a message to the user. Use `file("stdin")` whenever you want to specify "User input on console" as a data origin. This can be used instead of a file name in several data reading functions (`scan`, `read.table`, `read.csv`,...). After the user input is converted from strings to numbers, the plotting begins. There, it can be seen, that plotting commands which are meant to be written to a file must be enclosed in two commands. These are in this case `png(.)` and `dev.off()`. The first function depends on the desired output file format (other common choices being `jpeg(.)` and `pdf(.)`). The second function, `dev.off()` is always required. It writes the plot to the file and ends the plotting process.

Preparing a standalone R script

Linux/Mac

The standalone script's file must first be made executable. This can happen by right-clicking the file, opening "Properties" in the opening menu and checking the "Executable" checkbox in the "Permissions" tab. Alternatively, the command

```
chmod +x PATH/TO/SCRIPT/SCRIPTNAME.R
```

can be called in a Terminal.

Windows

For each standalone script, a batch file must be written with the following contents:

```
"C:\Program Files\R-XXXXXXX\bin\Rscript.exe" "%~dp0\XXXXXXX.R" %*
```

A batch file is a normal text file, but which has a `*.bat` extension except a `*.txt` extension. Create it using a text editor like `notepad` (not `Word`) or similar and put the file name into quotation marks `"FILENAME.bat"`) in the save dialog. To edit an existing batch file, right-click on it and select "Edit".

You have to adapt the code shown above everywhere `XXX...` is written:

- Insert the correct folder where your R installation resides
- Insert the correct name of your script and place it into the same directory as this batch file.

Explanation of the elements in the code: The first part `"C:\...\Rscript.exe"` tells Windows where to find the `Rscript.exe` program. The second part `"%~dp0\XXX.R"` tells `Rscript` to execute the R script you've written which resides in the same folder as the batch file (`%~dp0` stands for the batch file

www.dbooks.org

folder). Finally, `%*` forwards any command line arguments you give to the batch file to the R script.

If you double-click on the batch file, the R script is executed. If you drag files on the batch file, the corresponding file names are given to the R script as command line arguments.

Using littler to execute R scripts

littler (pronounced *little r*) (cran) provides, besides other features, two possibilities to run R scripts from the command line with littler's `r` command (when one works with Linux or MacOS).

Installing littler

From R:

```
install.packages("littler")
```

The path of `r` is printed in the terminal, like

```
You could link to the 'r' binary installed in
'/home/*USER*/R/x86_64-pc-linux-gnu-library/3.4/littler/bin/r'
from '/usr/local/bin' in order to use 'r' for scripting.
```

To be able to call `r` from the system's command line, a symlink is needed:

```
ln -s /home/*USER*/R/x86_64-pc-linux-gnu-library/3.4/littler/bin/r /usr/local/bin/r
```

Using apt-get (Debian, Ubuntu):

```
sudo apt-get install littler
```

Using littler with standard .r scripts

With `r` from `littler` it is possible to execute standalone R scripts without any changes to the script. Example script:

```
# User message (\n = end the line)
cat("Input numbers, separated by space:\n")
# Read user input as one string (n=1 -> Read only one line)
input <- readLines(file('stdin'), n=1)
# Split the string at each space (\\s == any space)
input <- strsplit(input, "\\s")[[1]]
# convert the obtained vector of strings to numbers
input <- as.numeric(input)

# Open the output picture file
png("hist.png",width=400, height=300)
# Draw the histogram
hist(input)
# Close the output file
```

```
dev.off()
```

Note that no shebang is at the top of the scripts. When saved as for example `hist.r`, it is directly callable from the system command:

```
r hist.r
```

Using littler on *shebanged* scripts

It is also possible to create executable R scripts with littler, with the use of the shebang

```
#!/usr/bin/env r
```

at the top of the script. The corresponding R script has to be made executable with `chmod +X /path/to/script.r` and is directly callable from the system terminal.

Read Standardize analyses by writing standalone R scripts online: https://riptutorial.com/r/topic/9937/standardize-analyses-by-writing-standalone-r-scripts

www.dbooks.org

Chapter 113: String manipulation with stringi package

Remarks

To install package simply run:

```
install.packages("stringi")
```

to load it:

```
require("stringi")
```

Examples

Count pattern inside string

With fixed pattern

```
stri_count_fixed("babab", "b")
# [1] 3
stri_count_fixed("babab", "ba")
# [1] 2
stri_count_fixed("babab", "bab")
# [1] 1
```

Natively:

```
length(gregexpr("b","babab")[[1]])
# [1] 3
length(gregexpr("ba","babab")[[1]])
# [1] 2
length(gregexpr("bab","babab")[[1]])
# [1] 1
```

function is vectorized over string and pattern:

```
stri_count_fixed("babab", c("b","ba"))
# [1] 3 2
stri_count_fixed(c("babab","bbb","bca","abc"), c("b","ba"))
# [1] 3 0 1 0
```

A base R solution:

```
sapply(c("b","ba"),function(x)length(gregexpr(x,"babab")[[1]]))
# b ba
```

```
# 3  2
```

With regex

First example - find `a` and any character after

Second example - find `a` and any digit after

```
stri_count_regex("a1 b2 a3 b4 aa", "a.")
# [1] 3
stri_count_regex("a1 b2 a3 b4 aa", "a\\d")
# [1] 2
```

Duplicating strings

```
stri_dup("abc",3)
# [1] "abcabcabc"
```

A base R solution that does the same would look like this:

```
paste0(rep("abc",3),collapse = "")
# [1] "abcabcabc"
```

Paste vectors

```
stri_paste(LETTERS,"-", 1:13)
# [1] "A-1"  "B-2"  "C-3"  "D-4"  "E-5"  "F-6"  "G-7"  "H-8"  "I-9"  "J-10" "K-11" "L-12" "M-
13"
# [14] "N-1"  "O-2"  "P-3"  "Q-4"  "R-5"  "S-6"  "T-7"  "U-8"  "V-9"  "W-10" "X-11" "Y-12" "Z-
13"
```

Natively, we could do this in R via:

```
> paste(LETTERS,1:13,sep="-")
 #[1] "A-1"  "B-2"  "C-3"  "D-4"  "E-5"  "F-6"  "G-7"  "H-8"  "I-9"  "J-10" "K-11" "L-12" "M-
13"
 #[14] "N-1"  "O-2" "P-3"  "Q-4"  "R-5"  "S-6"  "T-7"  "U-8"  "V-9"  "W-10" "X-11" "Y-12" "Z-
13"
```

Splitting text by some fixed pattern

Split vector of texts using one pattern:

```
stri_split_fixed(c("To be or not to be.", "This is very short sentence."),", ")
# [[1]]
# [1] "To"  "be"  "or"  "not" "to"  "be."
#
# [[2]]
# [1] "This"      "is"        "very"      "short"     "sentence."
```


www.dbooks.org

Split one text using many patterns:

```
stri_split_fixed("Apples, oranges and pineaplles.",c(" ", ",", "s"))
# [[1]]
# [1] "Apples,"     "oranges"     "and"         "pineaplles."
#
# [[2]]
# [1] "Apples"                  " oranges and pineaplles."
#
# [[3]]
# [1] "Apple"          ", orange"       " and pineaplle" "."
```

Read String manipulation with stringi package online: https://riptutorial.com/r/topic/1670/string-manipulation-with-stringi-package

Chapter 114: strsplit function

Syntax

- strsplit(
- x
- split
- fixed = FALSE
- perl = FALSE
- useBytes = FALSE)

Examples

Introduction

`strsplit` is a useful function for breaking up a vector into an list on some character pattern. With typical R tools, the whole list can be reincorporated to a data.frame or part of the list might be used in a graphing exercise.

Here is a common usage of `strsplit`: break a character vector along a comma separator:

```
temp <- c("this,that,other", "hat,scarf,food", "woman,man,child")
# get a list split by commas
myList <- strsplit(temp, split=",")
# print myList
myList
[[1]]
[1] "this"  "that"  "other"

[[2]]
[1] "hat"   "scarf" "food"

[[3]]
[1] "woman" "man"   "child"
```

As hinted above, the split argument is not limited to characters, but may follow a pattern dictated by a regular expression. For example, temp2 is identical to temp above except that the separators have been altered for each item. We can take advantage of the fact that the split argument accepts regular expressions to alleviate the irregularity in the vector.

```
temp2 <- c("this, that, other", "hat,scarf ,food", "woman; man ; child")
myList2 <- strsplit(temp2, split=" ?[,;] ?")
myList2
[[1]]
[1] "this"  "that"  "other"

[[2]]
[1] "hat"   "scarf" "food"
```


www.dbooks.org

```
[[3]]
[1] "woman" "man"   "child"
```

Notes:

1. breaking down the regular expression syntax is out of scope for this example.
2. Sometimes matching regular expressions can slow down a process. As with many R functions that allow the use of regular expressions, the fixed argument is available to tell R to match on the split characters literally.

Read strsplit function online: https://riptutorial.com/r/topic/2762/strsplit-function

Chapter 115: Subsetting

Introduction

Given an R object, we may require separate analysis for one or more parts of the data contained in it. The process of obtaining these parts of the data from a given object is called `subsetting`.

Remarks

Missing values:

Missing values (`NA`s) used in subsetting with `[` return `NA` since a `NA` index

> picks an unknown element and so returns NA in the corresponding element..

The "default" type of `NA` is "logical" (`typeof(NA)`) which means that, as any "logical" vector used in subsetting, will be **recycled** to match the length of the subsetted object. So `x[NA]` is equivalent to `x[as.logical(NA)]` which is equivalent to `x[rep_len(as.logical(NA), length(x))]` and, consequently, it returns a missing value (`NA`) for each element of `x`. As an example:

```
x <- 1:3
x[NA]
## [1] NA NA NA
```

While indexing with "numeric"/"integer" `NA` picks a single `NA` element (for each `NA` in index):

```
x[as.integer(NA)]
## [1] NA

x[c(NA, 1, NA, NA)]
## [1] NA  1 NA NA
```

Subsetting out of bounds:

The `[` operator, with one argument passed, allows indices that are `> length(x)` and returns `NA` for atomic vectors or `NULL` for generic vectors. In contrast, with `[[` and when `[` is passed more arguments (i.e. subsetting out of bounds objects with `length(dim(x)) > 2`) an error is returned:

```
(1:3)[10]
## [1] NA
(1:3)[[10]]
## Error in (1:3)[[10]] : subscript out of bounds
as.matrix(1:3)[10]
## [1] NA
as.matrix(1:3)[, 10]
## Error in as.matrix(1:3)[, 10] : subscript out of bounds
list(1, 2, 3)[10]
## [[1]]
## NULL
```


www.dbooks.org

```
list(1, 2, 3)[[10]]
## Error in list(1, 2, 3)[[10]] : subscript out of bounds
```

The behaviour is the same when subsetting with "character" vectors, that are not matched in the "names" attribute of the object, too:

```
c(a = 1, b = 2)["c"]
## <NA>
##   NA
list(a = 1, b = 2)["c"]
## <NA>
## NULL
```

Help topics:

See `?Extract` for further information.

Examples

Atomic vectors

Atomic vectors (which excludes lists and expressions, which are also vectors) are subset using the `[` operator:

```
# create an example vector
v1 <- c("a", "b", "c", "d")

# select the third element
v1[3]
## [1] "c"
```

The `[` operator can also take a vector as the argument. For example, to select the first and third elements:

```
v1 <- c("a", "b", "c", "d")

v1[c(1, 3)]
## [1] "a" "c"
```

Some times we may require to omit a particular value from the vector. This can be achieved using a negative sign(`-`) before the index of that value. For example, to omit to omit the first value from v1, use `v1[-1]`. This can be extended to more than one value in a straight forward way. For example, `v1[-c(1,3)]`.

```
> v1[-1]
[1] "b" "c" "d"
> v1[-c(1,3)]
[1] "b" "d"
```

On some occasions, we would like to know, especially, when the length of the vector is large,

index of a particular value, if it exists:

```
> v1=="c"
[1] FALSE FALSE  TRUE FALSE
> which(v1=="c")
[1] 3
```

If the atomic vector has names (a `names` attribute), it can be subset using a character vector of names:

```
v <- 1:3
names(v) <- c("one", "two", "three")

v
##  one   two three
##    1     2     3

v["two"]
## two
##   2
```

The `[[` operator can also be used to index atomic vectors, with differences in that it accepts a indexing vector with a length of one and strips any names present:

```
v[[c(1, 2)]]
## Error in v[[c(1, 2)]] :
##  attempt to select more than one element in vectorIndex

v[["two"]]
## [1] 2
```

Vectors can also be subset using a logical vector. In contrast to subsetting with numeric and character vectors, the logical vector used to subset has to be equal to the length of the vector whose elements are extracted, so if a logical vector `y` is used to subset `x`, i.e. `x[y]`, if `length(y) < length(x)` then `y` will be recycled to match `length(x)`:

```
v[c(TRUE, FALSE, TRUE)]
##  one three
##    1     3

v[c(FALSE, TRUE)]  # recycled to 'c(FALSE, TRUE, FALSE)'
## two
##   2

v[TRUE]   # recycled to 'c(TRUE, TRUE, TRUE)'
##  one   two three
##    1     2     3

v[FALSE]   # handy to discard elements but save the vector's type and basic structure
## named integer(0)
```

Lists

A list can be subset with `[`:

www.dbooks.org

```
l1 <- list(c(1, 2, 3), 'two' = c("a", "b", "c"), list(10, 20))
l1
## [[1]]
## [1] 1 2 3
##
## $two
## [1] "a" "b" "c"
##
## [[3]]
## [[3]][[1]]
## [1] 10
##
## [[3]][[2]]
## [1] 20

l1[1]
## [[1]]
## [1] 1 2 3

l1['two']
## $two
## [1] "a" "b" "c"

l1[[2]]
## [1] "a" "b" "c"

l1[['two']]
## [1] "a" "b" "c"
```

Note the result of `l1[2]` is still a list, as the `[` operator selects elements of a list, returning a smaller list. The `[[` operator extracts list elements, returning an object of the type of the list element.

Elements can be indexed by number or a character string of the name (if it exists). Multiple elements can be selected with `[` by passing a vector of numbers or strings of names. Indexing with a vector of `length > 1` in `[` and `[[` returns a "list" with the specified elements and a recursive subset (if available), *respectively*:

```
l1[c(3, 1)]
## [[1]]
## [[1]][[1]]
## [1] 10
##
## [[1]][[2]]
## [1] 20
##
##
## [[2]]
## [1] 1 2 3
```

Compared to:

```
l1[[c(3, 1)]]
## [1] 10
```

which is equivalent to:

```
l1[[3]][[1]]
## [1] 10
```

The `$` operator allows you to select list elements solely by name, but unlike `[` and `[[`, does not require quotes. As an infix operator, `$` can only take a single name:

```
l1$two
## [1] "a" "b" "c"
```

Also, the `$` operator allows for partial matching by default:

```
l1$t
## [1] "a" "b" "c"
```

in contrast with `[[` where it needs to be specified whether partial matching is allowed:

```
l1[["t"]]
## NULL
l1[["t", exact = FALSE]]
## [1] "a" "b" "c"
```

Setting `options(warnPartialMatchDollar = TRUE)`, a "warning" is given when partial matching happens with `$`:

```
l1$t
## [1] "a" "b" "c"
## Warning message:
## In l1$t : partial match of 't' to 'two'
```

Matrices

For each dimension of an object, the `[` operator takes one argument. Vectors have one dimension and take one argument. Matrices and data frames have two dimensions and take two arguments, given as `[i, j]` where `i` is the row and `j` is the column. Indexing starts at 1.

```
## a sample matrix
mat <- matrix(1:6, nrow = 2, dimnames = list(c("row1", "row2"), c("col1", "col2", "col3")))

mat
#      col1 col2 col3
# row1    1    3    5
# row2    2    4    6
```

`mat[i,j]` is the element in the `i`-th row, `j`-th column of the matrix `mat`. For example, an `i` value of `2` and a `j` value of `1` gives the number in the second row and the first column of the matrix. Omitting `i` or `j` returns all values in that dimension.

```
mat[ , 3]
## row1 row2
##    5    6
```


www.dbooks.org

```
mat[1, ]
# col1 col2 col3
#    1    3    5
```

When the matrix has row or column names (not required), these can be used for subsetting:

```
mat[ , 'col1']
# row1 row2
#    1    2
```

By default, the result of a subset will be simplified if possible. If the subset only has one dimension, as in the examples above, the result will be a one-dimensional vector rather than a two-dimensional matrix. This default can be overriden with the `drop = FALSE` argument to `[`:

```
## This selects the first row as a vector
class(mat[1, ])
# [1] "integer"

## Whereas this selects the first row as a 1x3 matrix:
class(mat[1, , drop = F])
# [1] "matrix"
```

Of course, dimensions cannot be dropped if the selection itself has two dimensions:

```
mat[1:2, 2:3]  ## A 2x2 matrix
#      col2 col3
# row1    3    5
# row2    4    6
```

Selecting individual matrix entries by their positions

It is also possible to use a Nx2 matrix to select N individual elements from a matrix (like how a coordinate system works). If you wanted to extract, in a vector, the entries of a matrix in the `(1st row, 1st column), (1st row, 3rd column), (2nd row, 3rd column), (2nd row, 1st column)` this can be done easily by creating a index matrix with those coordinates and using that to subset the matrix:

```
mat
#      col1 col2 col3
# row1    1    3    5
# row2    2    4    6

ind = rbind(c(1, 1), c(1, 3), c(2, 3), c(2, 1))
ind
#      [,1] [,2]
# [1,]    1    1
# [2,]    1    3
# [3,]    2    3
# [4,]    2    1

mat[ind]
# [1] 1 5 6 2
```

In the above example, the 1st column of the `ind` matrix refers to rows in `mat`, the 2nd column of `ind` refers to columns in `mat`.

Data frames

Subsetting a data frame into a smaller data frame can be accomplished the same as subsetting a list.

```
> df3 <- data.frame(x = 1:3, y = c("a", "b", "c"), stringsAsFactors = FALSE)

> df3
##   x y
## 1 1 a
## 2 2 b
## 3 3 c

> df3[1]   # Subset a variable by number
##   x
## 1 1
## 2 2
## 3 3

> df3["x"]   # Subset a variable by name
##   x
## 1 1
## 2 2
## 3 3

> is.data.frame(df3[1])
## TRUE

> is.list(df3[1])
## TRUE
```

Subsetting a dataframe into a column vector can be accomplished using double brackets [[ ]] or the dollar sign operator $.

```
> df3[[2]]    # Subset a variable by number using [[ ]]
## [1] "a" "b" "c"

> df3[["y"]]  # Subset a variable by name using [[ ]]
## [1] "a" "b" "c"

> df3$x    # Subset a variable by name using $
## [1] 1 2 3

> typeof(df3$x)
## "integer"

> is.vector(df3$x)
## TRUE
```

Subsetting a data as a two dimensional matrix can be accomplished using `i` and `j` terms.

```
> df3[1, 2]    # Subset row and column by number
## [1] "a"
```


www.dbooks.org

```
> df3[1, "y"]  # Subset row by number and column by name
## [1] "a"

> df3[2, ]     # Subset entire row by number
##   x y
## 2 2 b

> df3[ , 1]    # Subset all first variables
## [1] 1 2 3

> df3[ , 1, drop = FALSE]
##   x
## 1 1
## 2 2
## 3 3
```

Note: Subsetting by `j` (column) alone simplifies to the variable's own type, but subsetting by `i` alone returns a `data.frame`, as the different variables may have different types and classes. Setting the `drop` parameter to `FALSE` keeps the data frame.

```
> is.vector(df3[, 2])
## TRUE

> is.data.frame(df3[2, ])
## TRUE

> is.data.frame(df3[, 2, drop = FALSE])
## TRUE
```

Other objects

The `[` and `[[` operators are primitive functions that are generic. This means that any *object* in R (specifically `isTRUE(is.object(x))` --i.e. has an explicit "class" attribute) can have its own specified behaviour when subsetted; i.e. has its own *methods* for `[` and/or `[[`.

For example, this is the case with "data.frame" (`is.object(iris)`) objects where `[.data.frame` and `[[.data.frame` methods are defined and they are made to exhibit both "matrix"-like and "list"-like subsetting. With forcing an error when subsetting a "data.frame", we see that, actually, a function `[.data.frame` was called when we -just- used `[`.

```
iris[invalidArgument, ]
## Error in `[.data.frame`(iris, invalidArgument, ) :
##   object 'invalidArgument' not found
```

Without further details on the current topic, an example `[` method:

```
x = structure(1:5, class = "myClass")
x[c(3, 2, 4)]
## [1] 3 2 4
'[.myClass' = function(x, i) cat(sprintf("We'd expect '%s[%s]' to be returned but this a
custom `[` method and should have a `?[.myClass` help page for its behaviour\n",
deparse(substitute(x)), deparse(substitute(i))))
```

```
x[c(3, 2, 4)]
## We'd expect 'x[c(3, 2, 4)]' to be returned but this a custom `[` method and should have a
`?[.myClass` help page for its behaviour
## NULL
```

We can overcome the method dispatching of `[` by using the equivalent non-generic `.subset` (and `.subset2` for `[[`). This is especially useful and efficient when programming our own "class"es and want to avoid work-arounds (like `unclass(x)`) when computing on our "class"es efficiently (avoiding method dispatch and copying objects):

```
.subset(x, c(3, 2, 4))
## [1] 3 2 4
```

Vector indexing

For this example, we will use the vector:

```
> x <- 11:20
> x
 [1] 11 12 13 14 15 16 17 18 19 20
```

R vectors are 1-indexed, so for example `x[1]` will return `11`. We can also extract a sub-vector of `x` by passing a vector of indices to the bracket operator:

```
> x[c(2,4,6)]
[1] 12 14 16
```

If we pass a vector of negative indices, R will return a sub-vector with the specified indices excluded:

```
> x[c(-1,-3)]
[1] 12 14 15 16 17 18 19 20
```

We can also pass a boolean vector to the bracket operator, in which case it returns a sub-vector corresponding to the coordinates where the indexing vector is `TRUE`:

```
> x[c(rep(TRUE,5),rep(FALSE,5))]
[1] 11 12 13 14 15 16
```

If the indexing vector is shorter than the length of the array, then it will be repeated, as in:

```
> x[c(TRUE,FALSE)]
[1] 11 13 15 17 19
> x[c(TRUE,FALSE,FALSE)]
[1] 11 14 17 20
```

Elementwise Matrix Operations

Let A and B be two matrices of same dimension. The operators `+,-,/,*,^` when used with matrices

www.dbooks.org

of same dimension perform the required operations on the corresponding elements of the matrices and return a new matrix of the same dimension. These operations are usually referred to as element-wise operations.

Operator	A op B	Meaning
+	A + B	Addition of corresponding elements of A and B
-	A - B	Subtracts the elements of B from the corresponding elements of A
/	A / B	Divides the elements of A by the corresponding elements of B
*	A * B	Multiplies the elements of A by the corresponding elements of B
^	A^(-1)	For example, gives a matrix whose elements are reciprocals of A

For "true" matrix multiplication, as seen in *Linear Algebra*, use `%*%`. For example, multiplication of A with B is: `A %*% B`. The dimensional requirements are that the `ncol()` of `A` be the same as `nrow()` of `B`

Some Functions used with Matrices

Function	Example	Purpose
nrow()	nrow(A)	determines the number of rows of A
ncol()	ncol(A)	determines the number of columns of A
rownames()	rownames(A)	prints out the row names of the matrix A
colnames()	colnames(A)	prints out the column names of the matrix A
rowMeans()	rowMeans(A)	computes means of each row of the matrix A
colMeans()	colMeans(A)	computes means of each column of the matrix A
upper.tri()	upper.tri(A)	returns a vector whose elements are the upper
		triangular matrix of square matrix A
lower.tri()	lower.tri(A)	returns a vector whose elements are the lower
		triangular matrix of square matrix A
det()	det(A)	results in the determinant of the matrix A
solve()	solve(A)	results in the inverse of the non-singular matrix A
diag()	diag(A)	returns a diagonal matrix whose off-diagnal elemts are zeros and

Function	Example	Purpose
		diagonals are the same as that of the square matrix A
t()	t(A)	returns the the transpose of the matrix A
eigen()	eigen(A)	retuens the eigenvalues and eigenvectors of the matrix A
is.matrix()	is.matrix(A)	returns TRUE or FALSE depending on whether A is a matrix or not.
as.matrix()	as.matrix(x)	creates a matrix out of the vector x

Read Subsetting online: https://riptutorial.com/r/topic/1686/subsetting

www.dbooks.org

Chapter 116: Survival analysis

Examples

Random Forest Survival Analysis with randomForestSRC

Just as the random forest algorithm may be applied to regression and classification tasks, it can also be extended to survival analysis.

In the example below a survival model is fit and used for prediction, scoring, and performance analysis using the package `randomForestSRC` from CRAN.

```
require(randomForestSRC)

set.seed(130948) #Other seeds give similar comparative results
x1   <- runif(1000)
y    <- rnorm(1000, mean = x1, sd = .3)
data <- data.frame(x1 = x1, y = y)
head(data)
```

```
         x1          y
1 0.9604353  1.3549648
2 0.3771234  0.2961592
3 0.7844242  0.6942191
4 0.9860443  1.5348900
5 0.1942237  0.4629535
6 0.7442532 -0.0672639
```

```
(modRFSRC <- rfsrc(y ~ x1, data = data, ntree=500, nodesize = 5))
```

```
                            Sample size: 1000
                     Number of trees: 500
           Minimum terminal node size: 5
       Average no. of terminal nodes: 208.258
No. of variables tried at each split: 1
              Total no. of variables: 1
                           Analysis: RF-R
                             Family: regr
                     Splitting rule: mse
                % variance explained: 32.08
                          Error rate: 0.11
```

```
x1new   <- runif(10000)
ynew    <- rnorm(10000, mean = x1new, sd = .3)
newdata <- data.frame(x1 = x1new, y = ynew)

survival.results <- predict(modRFSRC, newdata = newdata)
survival.results
```

```
  Sample size of test (predict) data: 10000
```

```
               Number of grow trees: 500
  Average no. of grow terminal nodes: 208.258
         Total no. of grow variables: 1
                             Analysis: RF-R
                               Family: regr
                 % variance explained: 34.97
                  Test set error rate: 0.11
```

Introduction - basic fitting and plotting of parametric survival models with the survival package

`survival` is the most commonly used package for survival analysis in R. Using the built-in `lung` dataset we can get started with Survival Analysis by fitting a regression model with the `survreg()` function, creating a curve with `survfit()`, and plotting predicted survival curves by calling the `predict` method for this package with new data.

In the example below we plot 2 predicted curves and vary `sex` between the 2 sets of new data, to visualize its effect:

```
require(survival)
s <- with(lung,Surv(time,status))

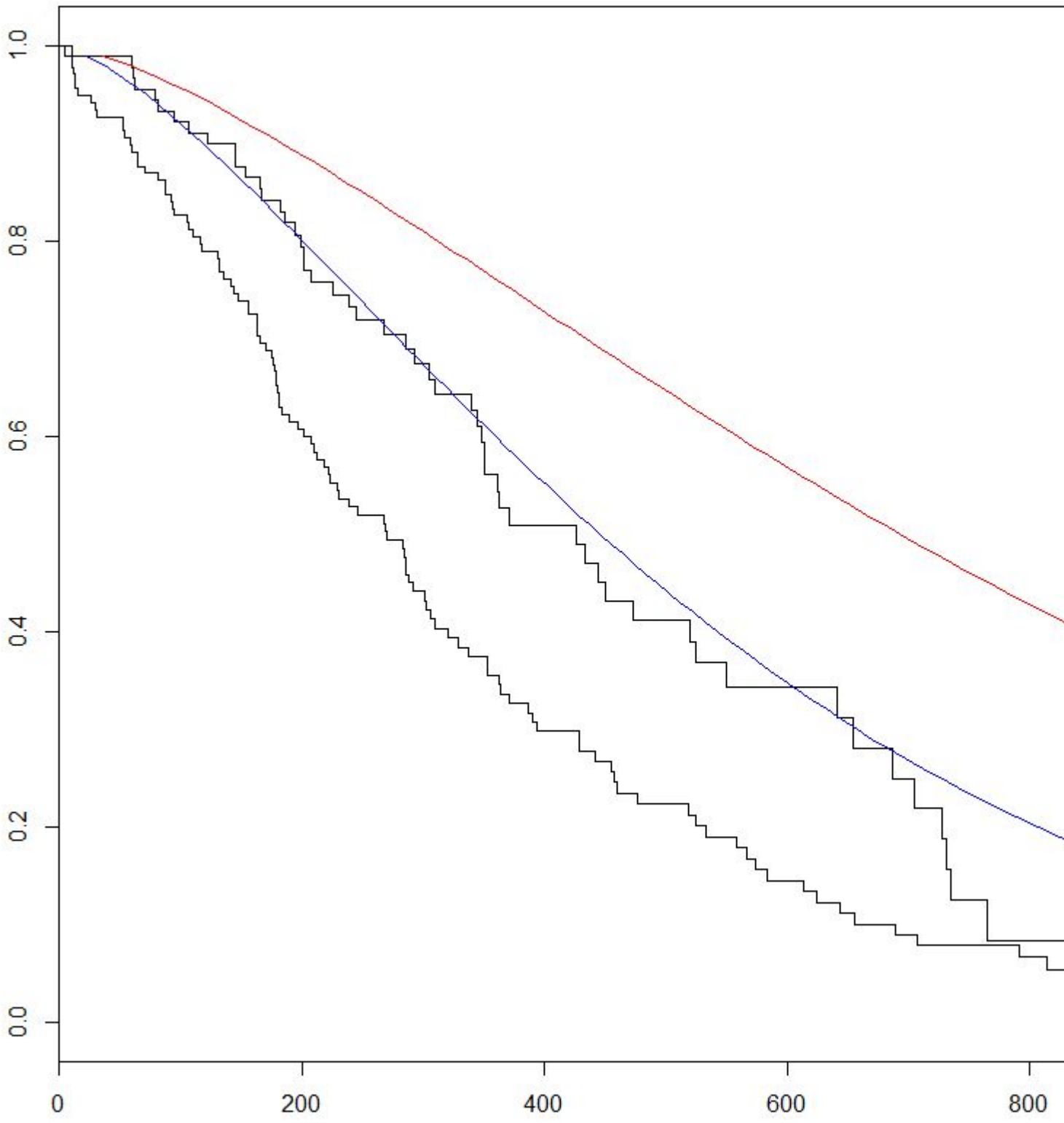
sWei  <- survreg(s ~ as.factor(sex)+age+ph.ecog+wt.loss+ph.karno,dist='weibull',data=lung)

fitKM <- survfit(s ~ sex,data=lung)
plot(fitKM)

lines(predict(sWei, newdata = list(sex      = 1,
                                   age      = 1,
                                   ph.ecog  = 1,
                                   ph.karno = 90,
                                   wt.loss  = 2),
                                 type = "quantile",
                                 p    = seq(.01, .99, by = .01)),
                                 seq(.99, .01, by        =-.01),
                                 col = "blue")

lines(predict(sWei, newdata = list(sex      = 2,
                                   age      = 1,
                                   ph.ecog  = 1,
                                   ph.karno = 90,
                                   wt.loss  = 2),
                                 type = "quantile",
                                 p    = seq(.01, .99, by = .01)),
                                 seq(.99, .01, by        =-.01),
                                 col = "red")
```


www.dbooks.org

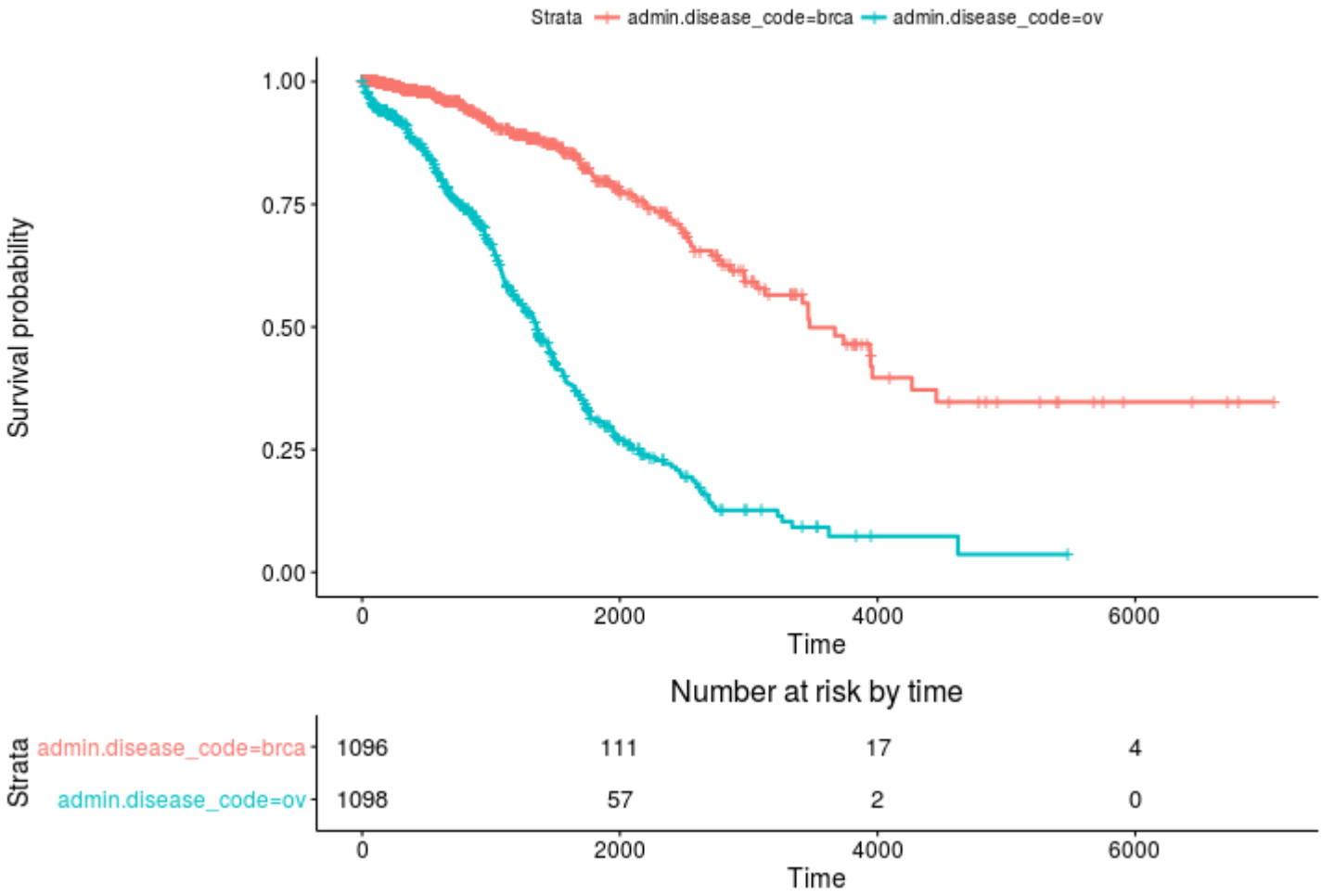


Kaplan Meier estimates of survival curves and risk set tables with survminer

Base plot

```
install.packages('survminer')
source("https://bioconductor.org/biocLite.R")
biocLite("RTCGA.clinical") # data for examples
```

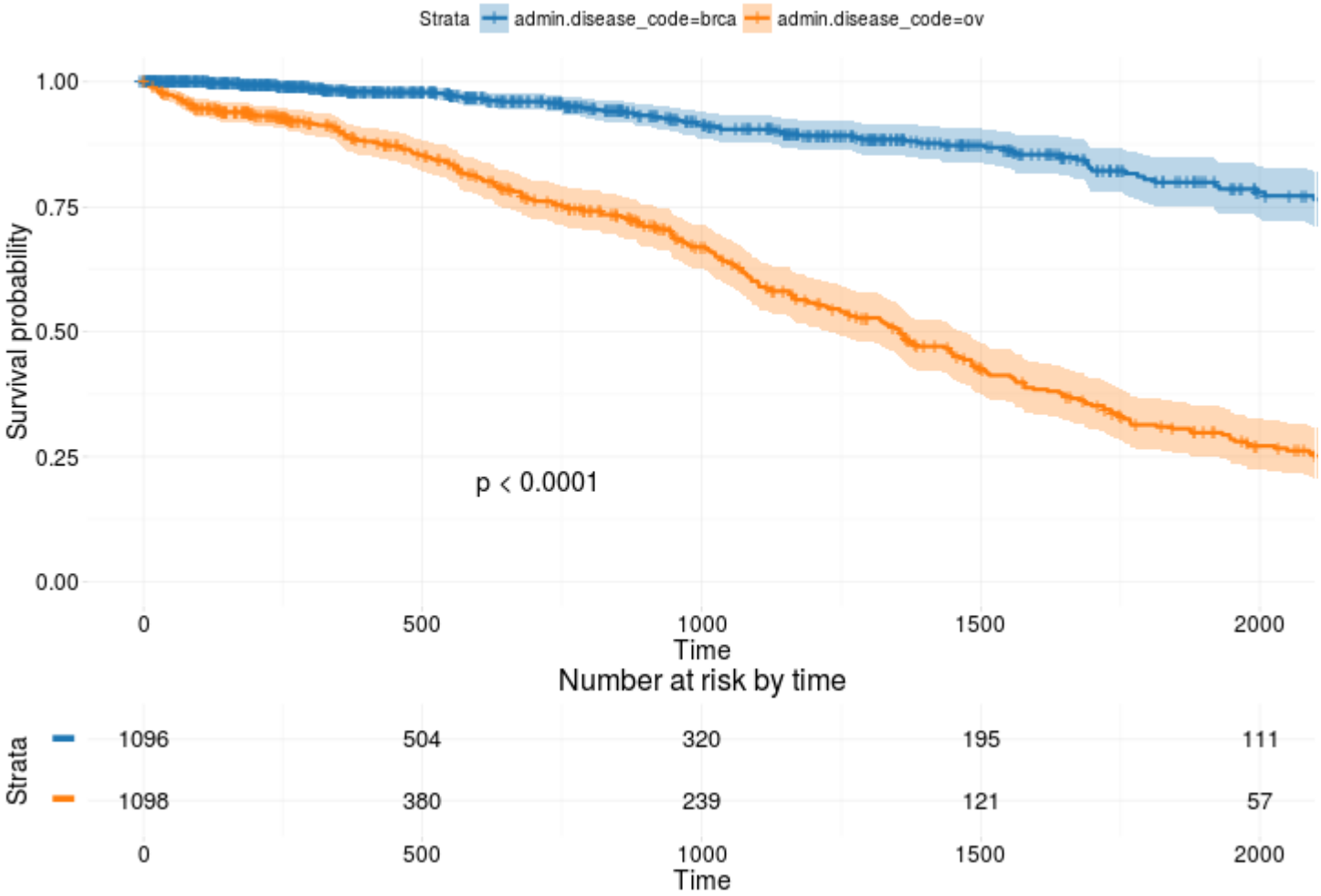
```
library(RTCGA.clinical)
survivalTCGA(BRCA.clinical, OV.clinical,
             extract.cols = "admin.disease_code") -> BRCAOV.survInfo
library(survival)
fit <- survfit(Surv(times, patient.vital_status) ~ admin.disease_code,
               data = BRCAOV.survInfo)
library(survminer)
ggsurvplot(fit, risk.table = TRUE)
```



More advanced

```
ggsurvplot(
   fit,                     # survfit object with calculated statistics.
   risk.table = TRUE,       # show risk table.
   pval = TRUE,             # show p-value of log-rank test.
   conf.int = TRUE,         # show confidence intervals for
                            # point estimaes of survival curves.
   xlim = c(0,2000),        # present narrower X axis, but not affect
                            # survival estimates.
   break.time.by = 500,     # break X axis in time intervals by 500.
   ggtheme = theme_RTCGA(), # customize plot and risk table with a theme.
 risk.table.y.text.col = T, # colour risk table text annotations.
  risk.table.y.text = FALSE # show bars instead of names in text annotations
                            # in legend of risk table
)
```


www.dbooks.org



Based on

http://r-addict.com/2016/05/23/Informative-Survival-Plots.html

Read Survival analysis online: https://riptutorial.com/r/topic/3788/survival-analysis

Chapter 117: Text mining

Examples

Scraping Data to build N-gram Word Clouds

The following example utilizes the `tm` text mining package to scrape and mine text data from the web to build word clouds with symbolic shading and ordering.

```
require(RWeka)
require(tau)
require(tm)
require(tm.plugin.webmining)
require(wordcloud)

# Scrape Google Finance ------------------------------------------------
googlefinance <- WebCorpus(GoogleFinanceSource("NASDAQ:LFVN"))

# Scrape Google News ---------------------------------------------------
lv.googlenews <- WebCorpus(GoogleNewsSource("LifeVantage"))
p.googlenews  <- WebCorpus(GoogleNewsSource("Protandim"))
ts.googlenews <- WebCorpus(GoogleNewsSource("TrueScience"))

# Scrape NYTimes -------------------------------------------------------
lv.nytimes <- WebCorpus(NYTimesSource(query = "LifeVantage", appid = nytimes_appid))
p.nytimes  <- WebCorpus(NYTimesSource("Protandim", appid = nytimes_appid))
ts.nytimes <- WebCorpus(NYTimesSource("TrueScience", appid = nytimes_appid))

# Scrape Reuters -------------------------------------------------------
lv.reutersnews <- WebCorpus(ReutersNewsSource("LifeVantage"))
p.reutersnews  <- WebCorpus(ReutersNewsSource("Protandim"))
ts.reutersnews <- WebCorpus(ReutersNewsSource("TrueScience"))

# Scrape Yahoo! Finance ------------------------------------------------
lv.yahoofinance <- WebCorpus(YahooFinanceSource("LFVN"))

# Scrape Yahoo! News ---------------------------------------------------
lv.yahoonews <- WebCorpus(YahooNewsSource("LifeVantage"))
p.yahoonews  <- WebCorpus(YahooNewsSource("Protandim"))
ts.yahoonews <- WebCorpus(YahooNewsSource("TrueScience"))

# Scrape Yahoo! Inplay -------------------------------------------------
lv.yahooinplay <- WebCorpus(YahooInplaySource("LifeVantage"))

# Text Mining the Results ----------------------------------------------
corpus <- c(googlefinance, lv.googlenews, p.googlenews, ts.googlenews, lv.yahoofinance,
lv.yahoonews, p.yahoonews,
ts.yahoonews, lv.yahooinplay) #lv.nytimes, p.nytimes, ts.nytimes,lv.reutersnews,
p.reutersnews, ts.reutersnews,

inspect(corpus)
wordlist <- c("lfvn", "lifevantage", "protandim", "truescience", "company", "fiscal",
"nasdaq")

ds0.1g <- tm_map(corpus, content_transformer(tolower))
ds1.1g <- tm_map(ds0.1g, content_transformer(removeWords), wordlist)
```


www.dbooks.org

```
ds1.1g <- tm_map(ds1.1g, content_transformer(removeWords), stopwords("english"))
ds2.1g <- tm_map(ds1.1g, stripWhitespace)
ds3.1g <- tm_map(ds2.1g, removePunctuation)
ds4.1g <- tm_map(ds3.1g, stemDocument)

tdm.1g <- TermDocumentMatrix(ds4.1g)
dtm.1g <- DocumentTermMatrix(ds4.1g)

findFreqTerms(tdm.1g, 40)
findFreqTerms(tdm.1g, 60)
findFreqTerms(tdm.1g, 80)
findFreqTerms(tdm.1g, 100)

findAssocs(dtm.1g, "skin", .75)
findAssocs(dtm.1g, "scienc", .5)
findAssocs(dtm.1g, "product", .75)

tdm89.1g <- removeSparseTerms(tdm.1g, 0.89)
tdm9.1g  <- removeSparseTerms(tdm.1g, 0.9)
tdm91.1g <- removeSparseTerms(tdm.1g, 0.91)
tdm92.1g <- removeSparseTerms(tdm.1g, 0.92)

tdm2.1g <- tdm92.1g

# Creates a Boolean matrix (counts # docs w/terms, not raw # terms)
tdm3.1g <- inspect(tdm2.1g)
tdm3.1g[tdm3.1g>=1] <- 1

# Transform into a term-term adjacency matrix
termMatrix.1gram <- tdm3.1g %*% t(tdm3.1g)

# inspect terms numbered 5 to 10
termMatrix.1gram[5:10,5:10]
termMatrix.1gram[1:10,1:10]

# Create a WordCloud to Visualize the Text Data ----------------------------
notsparse <- tdm2.1g
m = as.matrix(notsparse)
v = sort(rowSums(m),decreasing=TRUE)
d = data.frame(word = names(v),freq=v)

# Create the word cloud
pal = brewer.pal(9,"BuPu")
wordcloud(words = d$word,
          freq = d$freq,
          scale = c(3,.8),
          random.order = F,
          colors = pal)
```



Note the use of `random.order` and a sequential pallet from RColorBrewer, which allows the programmer to capture more information in the cloud by assigning meaning to the order and coloring of terms.

Above is the 1-gram case.

We can make a major leap to n-gram word clouds and in doing so we'll see how to make almost any text-mining analysis flexible enough to handle n-grams by transforming our TDM.

The initial difficulty you run into with n-grams in R is that `tm`, the most popular package for text mining, does not inherently support tokenization of bi-grams or n-grams. Tokenization is the process of representing a word, part of a word, or group of words (or symbols) as a single data element called a token.

Fortunately, we have some hacks which allow us to continue using `tm` with an upgraded tokenizer. There's more than one way to achieve this. We can write our own simple tokenizer using the `textcnt()` function from tau:

```
tokenize_ngrams <- function(x, n=3)
return(rownames(as.data.frame(unclass(textcnt(x,method="string",n=n)))))
```

or we can invoke `RWeka`'s tokenizer within `tm`:

```
# BigramTokenize
BigramTokenizer <- function(x) NGramTokenizer(x, Weka_control(min = 2, max = 2))
```

From this point you can proceed much as in the 1-gram case:

```
# Create an n-gram Word Cloud ------------------------------------------------
tdm.ng <- TermDocumentMatrix(ds5.1g, control = list(tokenize = BigramTokenizer))
```


www.dbooks.org

```
dtm.ng <- DocumentTermMatrix(ds5.1g, control = list(tokenize = BigramTokenizer))

# Try removing sparse terms at a few different levels
tdm89.ng <- removeSparseTerms(tdm.ng, 0.89)
tdm9.ng  <- removeSparseTerms(tdm.ng, 0.9)
tdm91.ng <- removeSparseTerms(tdm.ng, 0.91)
tdm92.ng <- removeSparseTerms(tdm.ng, 0.92)

notsparse <- tdm91.ng
m = as.matrix(notsparse)
v = sort(rowSums(m),decreasing=TRUE)
d = data.frame(word = names(v),freq=v)

# Create the word cloud
pal = brewer.pal(9,"BuPu")
wordcloud(words = d$word,
          freq = d$freq,
          scale = c(3,.8),
          random.order = F,
          colors = pal)
```



The example above is reproduced with permission from Hack-R's data science blog. Additional commentary may be found in the original article.

Read Text mining online: https://riptutorial.com/r/topic/3579/text-mining

Chapter 118: The character class

Introduction

Characters are what other languages call 'string vectors.'

Remarks

Related topics

Patterns

- Regular Expressions (regex)
- Pattern Matching and Replacement
- strsplit function

Input and output

- Reading and writing strings

Examples

Coercion

To check whether a value is a character use the `is.character()` function. To coerce a variable to a character use the `as.character()` function.

```
x <- "The quick brown fox jumps over the lazy dog"
class(x)
[1] "character"
is.character(x)
[1] TRUE
```

Note that numerics can be coerced to characters, but attempting to coerce a character to numeric may result in `NA`.

```
as.numeric("2")
[1] 2
as.numeric("fox")
[1] NA
Warning message:
NAs introduced by coercion
```

Read The character class online: https://riptutorial.com/r/topic/9017/the-character-class

www.dbooks.org

Chapter 119: The Date class

Remarks

Related topics

- Date and Time

Jumbled notes

- `Date`: Stores time as number of days since UNIX epoch on `1970-01-01`. with negative values for earlier dates.
- It is represented as an integer (however, it is not enforced in the internal representation)
- They are always printed following the rules of the current Gregorian calendar, even though the calendar was not in use a long time ago.
- It doesn't keep track of timezones, so it should not be used to truncate the time out of `POSIXct` or `POSIXlt` objects.
- `sys.Date()` returns an object of class `Date`

More notes

- lubridate's `ymd`, `mdy`, etc. are alternatives to `as.Date` that also parse to Date class; see Parsing dates and datetimes from strings with lubridate.
- data.table's experimental IDate class is derived from and is mostly interchangeable with Date, but is stored as integer instead of double.

Examples

Formatting Dates

To format `Dates` we use the `format(date, format="%Y-%m-%d")` function with either the `POSIXct` (given from `as.POSIXct()`) or `POSIXlt` (given from `as.POSIXlt()`)

```
d = as.Date("2016-07-21") # Current Date Time Stamp

format(d,"%a")            # Abbreviated Weekday
## [1] "Thu"

format(d,"%A")            # Full Weekday
## [1] "Thursday"

format(d,"%b")            # Abbreviated Month
## [1] "Jul"
```

```
format(d,"%B")            # Full Month
## [1] "July"

format(d,"%m")            # 00-12 Month Format
## [1] "07"

format(d,"%d")            # 00-31 Day Format
## [1] "21"

format(d,"%e")            # 0-31 Day Format
## [1] "21"

format(d,"%y")            # 00-99 Year
## [1] "16"

format(d,"%Y")            # Year with Century
## [1] "2016"
```

For more, see `?strptime`.

Dates

To coerce a variable to a date use the `as.Date()` function.

```
> x <- as.Date("2016-8-23")
> x
[1] "2016-08-23"
> class(x)
[1] "Date"
```

The `as.Date()` function allows you to provide a format argument. The default is `%Y-%m-%d`, which is Year-month-day.

```
> as.Date("23-8-2016", format="%d-%m-%Y") # To read in an European-style date
[1] "2016-08-23"
```

The format string can be placed either within a pair of single quotes or double quotes. Dates are usually expressed in a variety of forms such as: `"d-m-yy"` or `"d-m-YYYY"` or `"m-d-yy"` or `"m-d-YYYY"` or `"YYYY-m-d"` or `"YYYY-d-m"`. These formats can also be expressed by replacing `"-"` by `"/"`. Furher, dates are also expressed in the forms, say, "Nov 6, 1986" or "November 6, 1986" or "6 Nov, 1986" or "6 November, 1986" and so on. The **as.Date()** function accepts all such character strings and when we mention the appropriate format of the string, it always outputs the date in the form `"YYYY-m-d"`.

Suppose we have a date string `"9-6-1962"` in the format `"%d-%m-%Y"`.

```
#
# It tries to interprets the string as YYYY-m-d
#
> as.Date("9-6-1962")
[1] "0009-06-19"       #interprets as "%Y-%m-%d"
>
```


www.dbooks.org

```
as.Date("9/6/1962")
[1] "0009-06-19"       #again interprets as "%Y-%m-%d"
>
# It has no problem in understanding, if the date is in form  YYYY-m-d or YYYY/m/d
#
> as.Date("1962-6-9")
[1] "1962-06-09"        # no problem
> as.Date("1962/6/9")
[1] "1962-06-09"        # no problem
>
```

By specifying the correct format of the input string, we can get the desired results. We use the following codes for specifying the formats to the **as.Date()** function.

Format Code	Meaning
`%d`	day
`%m`	month
`%y`	year in 2-digits
`%Y`	year in 4-digits
`%b`	abbreviated month in 3 chars
`%B`	full name of the month

Consider the following example specifying the **format** parameter:

```
> as.Date("9-6-1962",format="%d-%m-%Y")
[1] "1962-06-09"
>
```

The parameter name **format** can be omitted.

```
> as.Date("9-6-1962", "%d-%m-%Y")
[1] "1962-06-09"
>
```

Some times, names of the months abbreviated to the first three characters are used in the writing the dates. In which case we use the format specifier `%b`.

```
> as.Date("6Nov1962","%d%b%Y")
[1] "1962-11-06"
>
```

Note that, there are no either `'-'` or `'/'` or white spaces between the members in the date string. The format string should exactly match that input string. Consider the following example:

```
> as.Date("6 Nov, 1962","%d %b, %Y")
```

```
[1] "1962-11-06"
>
```

Note that, there is a comma in the date string and hence a comma in the format specification too. If comma is omitted in the format string, it results in an `NA`. An example usage of `%B` format specifier is as follows:

```
> as.Date("October 12, 2016", "%B %d, %Y")
[1] "2016-10-12"
>
> as.Date("12 October, 2016", "%d %B, %Y")
[1] "2016-10-12"
>
```

`%y` format is system specific and hence, should be used with caution. Other parameters used with this function are **origin** and **tz**(time zone).

Parsing Strings into Date Objects

R contains a Date class, which is created with `as.Date()`, which takes a string or vector of strings, and if the date is not in ISO 8601 date format `YYYY-MM-DD`, a formatting string of `strptime`-style tokens.

```
as.Date('2016-08-01')    # in ISO format, so does not require formatting string
## [1] "2016-08-01"

as.Date('05/23/16', format = '%m/%d/%y')
## [1] "2016-05-23"

as.Date('March 23rd, 2016', '%B %drd, %Y')    # add separators and literals to format
## [1] "2016-03-23"

as.Date('  2016-08-01  foo')    # leading whitespace and all trailing characters are ignored
## [1] "2016-08-01"

as.Date(c('2016-01-01', '2016-01-02'))
# [1] "2016-01-01" "2016-01-02"
```

Read The Date class online: https://riptutorial.com/r/topic/9015/the-date-class

www.dbooks.org

Chapter 120: The logical class

Introduction

Logical is a mode (and an implicit class) for vectors.

Remarks

Shorthand

`TRUE`, `FALSE` and `NA` are the only values for logical vectors; and all three are reserved words. `T` and `F` can be shorthand for `TRUE` and `FALSE` in a clean R session, but neither `T` nor `F` are reserved, so assignment of non-default values to those names can set users up for difficulties.

Examples

Logical operators

There are two sorts of logical operators: those that accept and return vectors of any length (elementwise operators: `!`, `|`, `&`, `xor()`) and those that only evaluate the first element in each argument (`&&`, `||`). The second sort is primarily used as the `cond` argument to the `if` function.

Logical Operator	Meaning	Syntax
!	Not	!x
&	element-wise (vectorized) and	x & y
&&	and (single element only)	x && y
|	element-wise (vectorized) or	x | y
||	or (single element only)	x || y
xor	element-wise (vectorized) exclusive OR	xor(x,y)

Note that the `||` operator evaluates the left condition and if the left condition is TRUE the right side is never evaluated. This can save time if the first is the result of a complex operation. The `&&` operator will likewise return FALSE without evaluation of the second argument when the first element of the first argument is FALSE.

```
> x <- 5
> x > 6 || stop("X is too small")
```

```
Error: X is too small
> x > 3 || stop("X is too small")
[1] TRUE
```

To check whether a value is a logical you can use the `is.logical()` function.

Coercion

To coerce a variable to a logical use the `as.logical()` function.

```
> x <- 2
> z <- x > 4
> z
[1] FALSE
> class(x)
[1] "numeric"
> as.logical(2)
[1] TRUE
```

When applying `as.numeric()` to a logical, a double will be returned. `NA` is a logical value and a logical operator with an `NA` will return `NA` if the outcome is ambiguous.

Interpretation of NAs

See Missing values for details.

```
> TRUE & NA
[1] NA
> FALSE & NA
[1] FALSE
> TRUE || NA
[1] TRUE
> FALSE || NA
[1] NA
```

Read The logical class online: https://riptutorial.com/r/topic/9016/the-logical-class

www.dbooks.org

Chapter 121: tidyverse

Examples

Creating tbl_df's

A tbl_df (pronounced *tibble diff*) is a variation of a data frame that is often used in tidyverse packages. It is implemented in the tibble package.

Use the `as_data_frame` function to turn a data frame into a tbl_df:

```
library(tibble)
mtcars_tbl <- as_data_frame(mtcars)
```

One of the most notable differences between data.frames and tbl_dfs is how they print:

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
*  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1   21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
2   21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
3   22.8     4 108.0    93  3.85 2.320 18.61     1     1     4     1
4   21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
5   18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
6   18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
7   14.3     8 360.0   245  3.21 3.570 15.84     0     0     3     4
8   24.4     4 146.7    62  3.69 3.190 20.00     1     0     4     2
9   22.8     4 140.8    95  3.92 3.150 22.90     1     0     4     2
10  19.2     6 167.6   123  3.92 3.440 18.30     1     0     4     4
# ... with 22 more rows
```

- The printed output includes a summary of the dimensions of the table (`32 x 11`)
- It includes the type of each column (`dbl`)
- It prints a limited number of rows. (To change this use `options(tibble.print_max = [number])`).

Many functions in the dplyr package work naturally with tbl_dfs, such as `group_by()`.

tidyverse: an overview

What is `tidyverse`?

`tidyverse` is the fast and elegant way to turn basic `R` into an enhanced tool, redesigned by Hadley/Rstudio. The development of all packages included in `tidyverse` follow the principle rules of The tidy tools manifesto. But first, let the authors describe their masterpiece:

> The tidyverse is a set of packages that work in harmony because they share common data representations and API design. The tidyverse package is designed to make it easy to install and load core packages from the tidyverse in a single command.

> The best place to learn about all the packages in the tidyverse and how they fit together is R for Data Science. Expect to hear more about the tidyverse in the coming months as I work on improved package websites, making citation easier, and providing a common home for discussions about data analysis with the tidyverse.
>
> (source))

How to use it?

Just with the ordinary `R` packages, you need to install and load the package.

```
install.package("tidyverse")
library("tidyverse")
```

The difference is, on a single command a couple of dozens of packages are installed/loaded. As a bonus, one may rest assured that all the installed/loaded packages are of compatible versions.

What are those packages?

The commonly known and widely used packages:

- ggplot2: advanced data visualisation SO_doc
- dplyr: fast (Rcpp) and coherent approach to data manipulation SO_doc
- tidyr: tools for data tidying SO_doc
- readr: for data import.
- purrr: makes your pure functions purr by completing R's functional programming tools with important features from other languages, in the style of the JS packages underscore.js, lodash and lazy.js.
- tibble: a modern re-imagining of data frames.
- magrittr: piping to make code more readable SO_doc

Packages for manipulating specific data formats:

- hms: easily read times
- stringr: provide a cohesive set of functions designed to make working with strings as easy as posssible
- lubridate: advanced date/times manipulations SO_doc
- forcats: advanced work with factors.

Data import:

- DBI: defines a common interface between the R and database management systems (DBMS)
- haven: easily import SPSS, SAS and Stata files SO_doc
- httr: the aim of httr is to provide a wrapper for the curl package, customised to the demands of modern web APIs

- jsonlite: a fast JSON parser and generator optimized for statistical data and the web
- readxl: read.xls and .xlsx files without need for dependency packages SO_doc
- rvest: rvest helps you scrape information from web pages SO_doc
- xml2: for XML

And modelling:

- modelr: provides functions that help you create elegant pipelines when modelling
- broom: easily extract the models into tidy data

Finally, `tidyverse` suggest the use of:

- knitr: the amazing general-purpose literate programming engine, with lightweight API's designed to give users full control of the output without heavy coding work. SO_docs: one, two
- rmarkdown: Rstudio's package for reproducible programming. SO_docs: one, two, three, four

Read tidyverse online: https://riptutorial.com/r/topic/1395/tidyverse

Chapter 122: Time Series and Forecasting

Remarks

Forecasting and time-series analysis may be handled with commonplace functions from the `stats` package, such as `glm()` or a large number of specialized packages. The CRAN Task View for time-series analysis provides a detailed listing of key packages by topic with short descriptions.

Examples

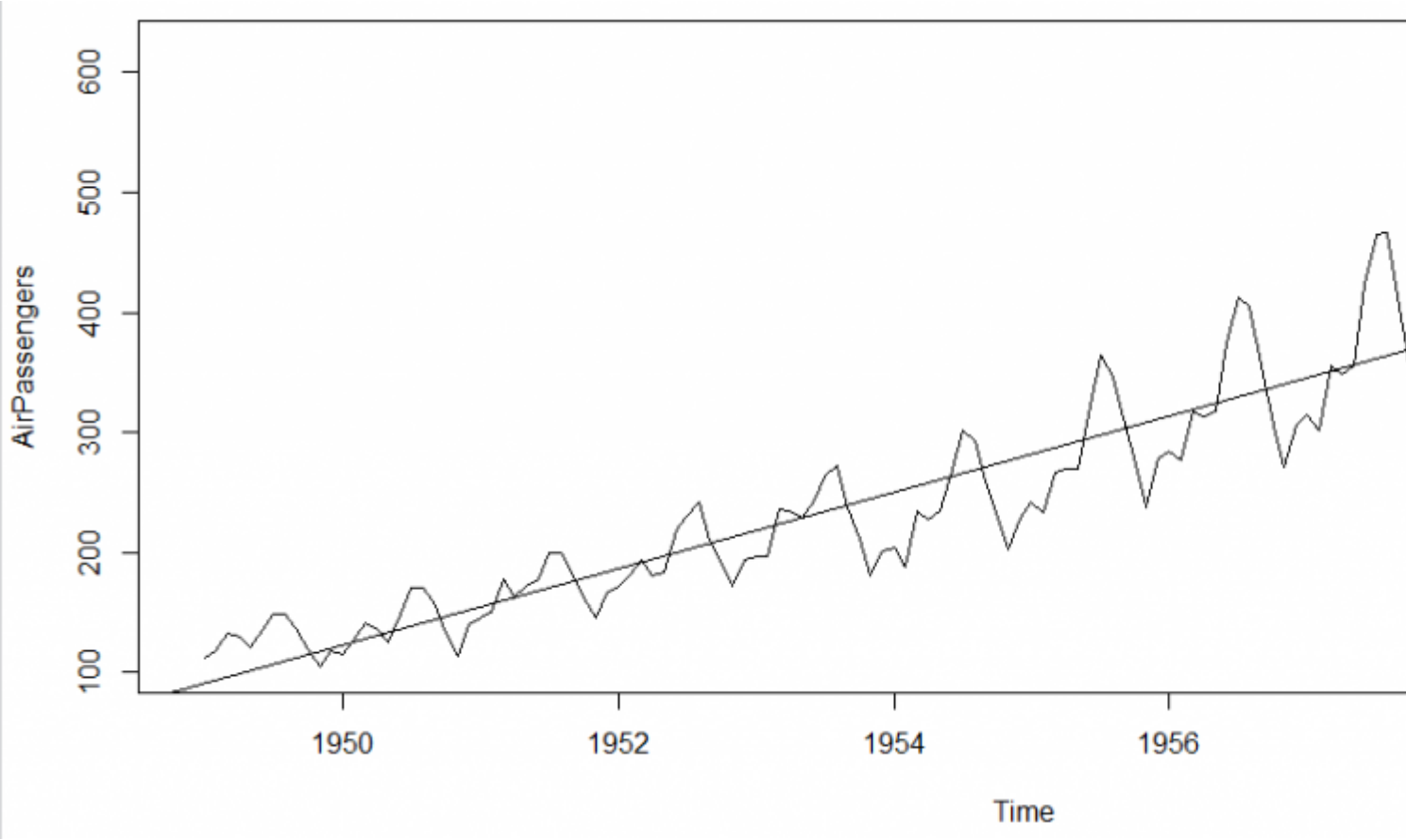
Exploratory Data Analysis with time-series data

```
data(AirPassengers)
class(AirPassengers)
```

```
1 "ts"
```

In the spirit of Exploratory Data Analysis (EDA) a good first step is to look at a plot of your time-series data:

```
plot(AirPassengers) # plot the raw data
abline(reg=lm(AirPassengers~time(AirPassengers))) # fit a trend line
```



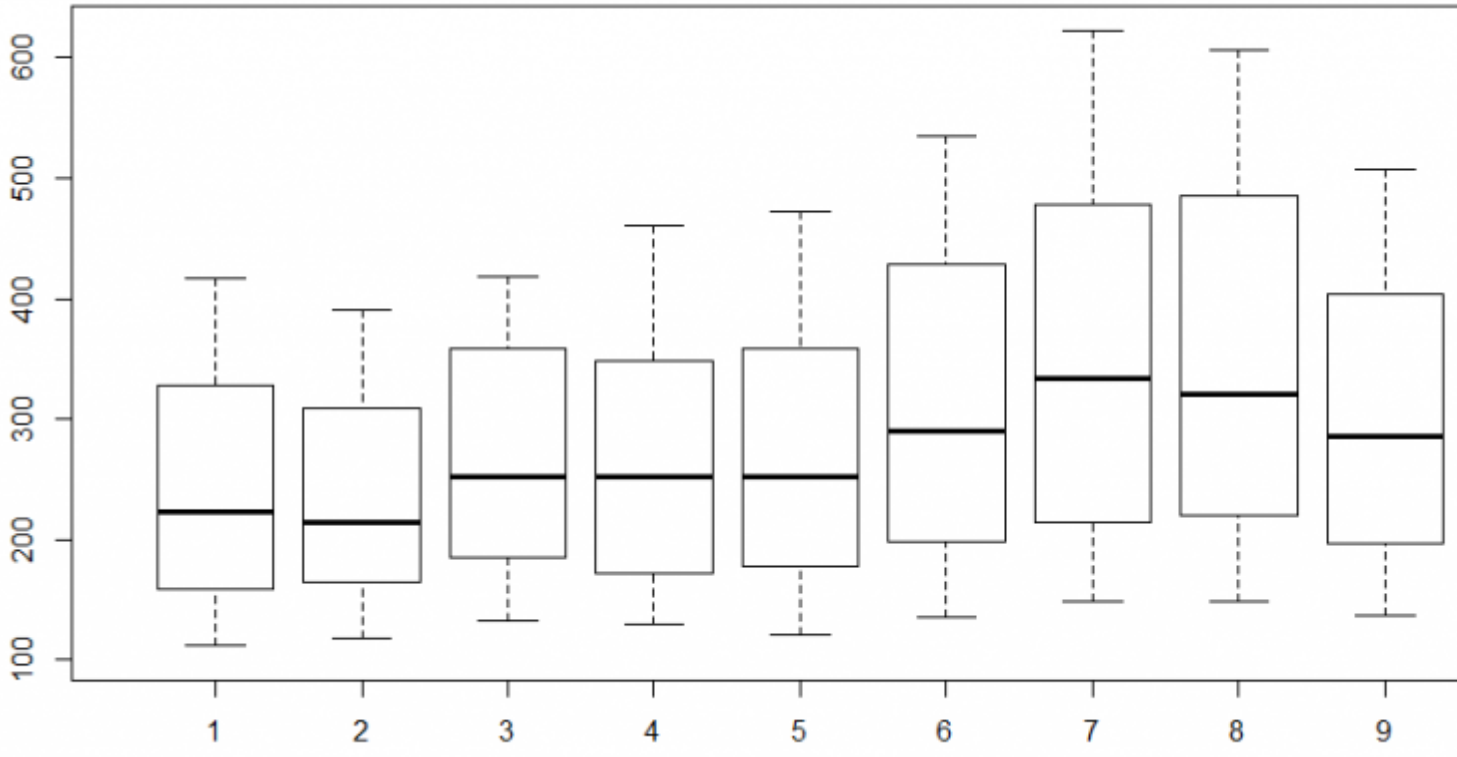
www.dbooks.org

For further EDA we examine cycles across years:

```
cycle(AirPassengers)
```

```
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949   1   2   3   4   5   6   7   8   9  10  11  12
1950   1   2   3   4   5   6   7   8   9  10  11  12
1951   1   2   3   4   5   6   7   8   9  10  11  12
1952   1   2   3   4   5   6   7   8   9  10  11  12
1953   1   2   3   4   5   6   7   8   9  10  11  12
1954   1   2   3   4   5   6   7   8   9  10  11  12
1955   1   2   3   4   5   6   7   8   9  10  11  12
1956   1   2   3   4   5   6   7   8   9  10  11  12
1957   1   2   3   4   5   6   7   8   9  10  11  12
1958   1   2   3   4   5   6   7   8   9  10  11  12
1959   1   2   3   4   5   6   7   8   9  10  11  12
1960   1   2   3   4   5   6   7   8   9  10  11  12
```

```
boxplot(AirPassengers~cycle(AirPassengers)) #Box plot across months to explore seasonal
effects
```



Creating a ts object

Time series data can be stored as a `ts` object. `ts` objects contain information about seasonal frequency that is used by ARIMA functions. It also allows for calling of elements in the series by date using the `window` command.

```
#Create a dummy dataset of 100 observations
x <- rnorm(100)
```

```
#Convert this vector to a ts object with 100 annual observations
x <- ts(x, start = c(1900), freq = 1)

#Convert this vector to a ts object with 100 monthly observations starting in July
x <- ts(x, start = c(1900, 7), freq = 12)

    #Alternatively, the starting observation can be a number:
    x <- ts(x, start = 1900.5, freq = 12)

#Convert this vector to a ts object with 100 daily observations and weekly frequency starting
in the first week of 1900
x <- ts(x, start = c(1900, 1), freq = 7)

#The default plot for a ts object is a line plot
plot(x)

#The window function can call elements or sets of elements by date

    #Call the first 4 weeks of 1900
    window(x, start = c(1900, 1), end = (1900, 4))

    #Call only the 10th week in 1900
    window(x, start = c(1900, 10), end = (1900, 10))

    #Call all weeks including and after the 10th week of 1900
    window(x, start = c(1900, 10))
```

It is possible to create `ts` objects with multiple series:

```
#Create a dummy matrix of 3 series with 100 observations each
x <- cbind(rnorm(100), rnorm(100), rnorm(100))

#Create a multi-series ts with annual observation starting in 1900
x <- ts(x, start = 1900, freq = 1)

#R will draw a plot for each series in the object
plot(x)
```

Read Time Series and Forecasting online: https://riptutorial.com/r/topic/2701/time-series-and-forecasting

www.dbooks.org

Chapter 123: Updating R and the package library

Examples

On Windows

Default installation of R on Windows stored files (and thus library) on a dedicated folder per R version on program files.

That means that by default, you would work with several versions of R in parallel and thus separate libraries.

If this not what you want and you prefer to always work with a single R instance you wan't to gradually update, it is recommended to modify the R installation folder. In wizard, just specify this folder (I personally use `c:\stats\R`). Then, for any upgrade, one possibility is to overwrite this R. Whether you also want to upgrade (all) packages is a delicate choice as it may break some of your code (this appeared for me with the `tm`package). You may:

- First make a copy of all your library before upgrading packages
- Maintain your own source packages repository, for instance using package `miniCRAN`

If you want to upgrade all packages - without any check, you can call use `packageStatus` as in:

```
pkgs <- packageStatus()  # choose mirror
upgrade(pkgs)
```

Finally, there exists a very convenient package to perform all operations, namely `installr`, even coming with a dedicated gui. If you want to use gui, you must use Rgui and not load the package in RStudio. Using the package with code is as simple as:

```
install.packages("installr") # install
setInternet2(TRUE) # only for R versions older than 3.3.0
installr::updateR() # updating R.
```

I refer to the great documentation https://www.r-statistics.com/tag/installr/ and specifically the step by step process with screenshots on Windows:https://www.r-statistics.com/2015/06/a-step-by-step-screenshots-tutorial-for-upgrading-r-on-windows/

Note that still I advocate using a single directory, ie. removing reference to the R version in installation folder name.

Read Updating R and the package library online: https://riptutorial.com/r/topic/4088/updating-r-and-the-package-library

Chapter 124: Updating R version

Introduction

Installing or Updating your Software will give access to new features and bug fixes. Updating your R installation can be done in a couple of ways. One Simple way is go to R website and download the latest version for your system.

Examples

Installing from R Website

To get the latest release go to https://cran.r-project.org/ and download the file for your operating system. Open the downloaded file and follow the on-screen installation steps. All the settings can be left on default unless you want to change a certain behaviour.

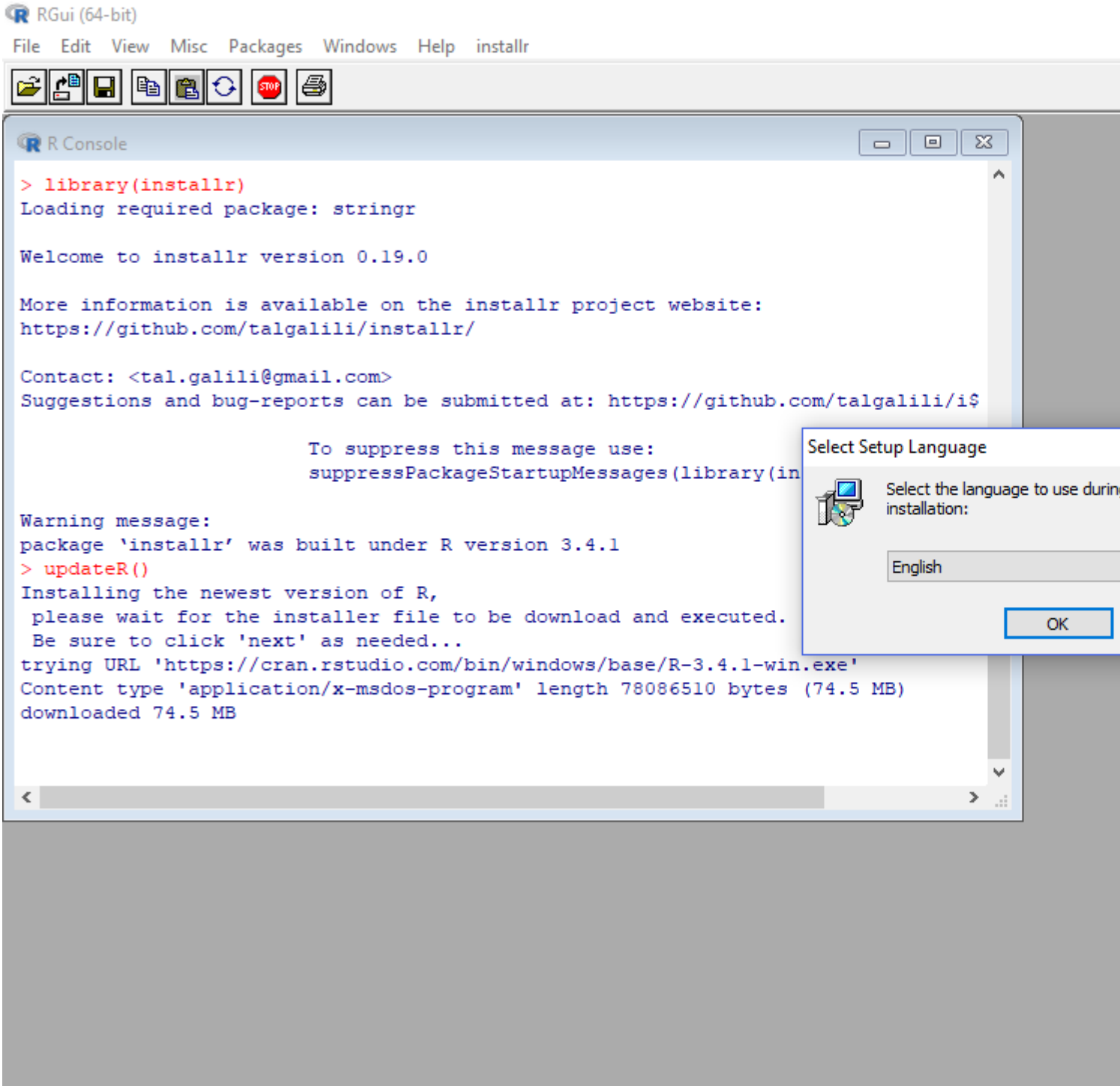
Updating from within R using installr Package

You can also update R from within R by using a handy package called **installr**.

Open R Console (NOT RStudio, this doesn't work from RStudio) and run the following code to install the package and initiate update.

```
install.packages("installr")
library("installr")
updateR()
```

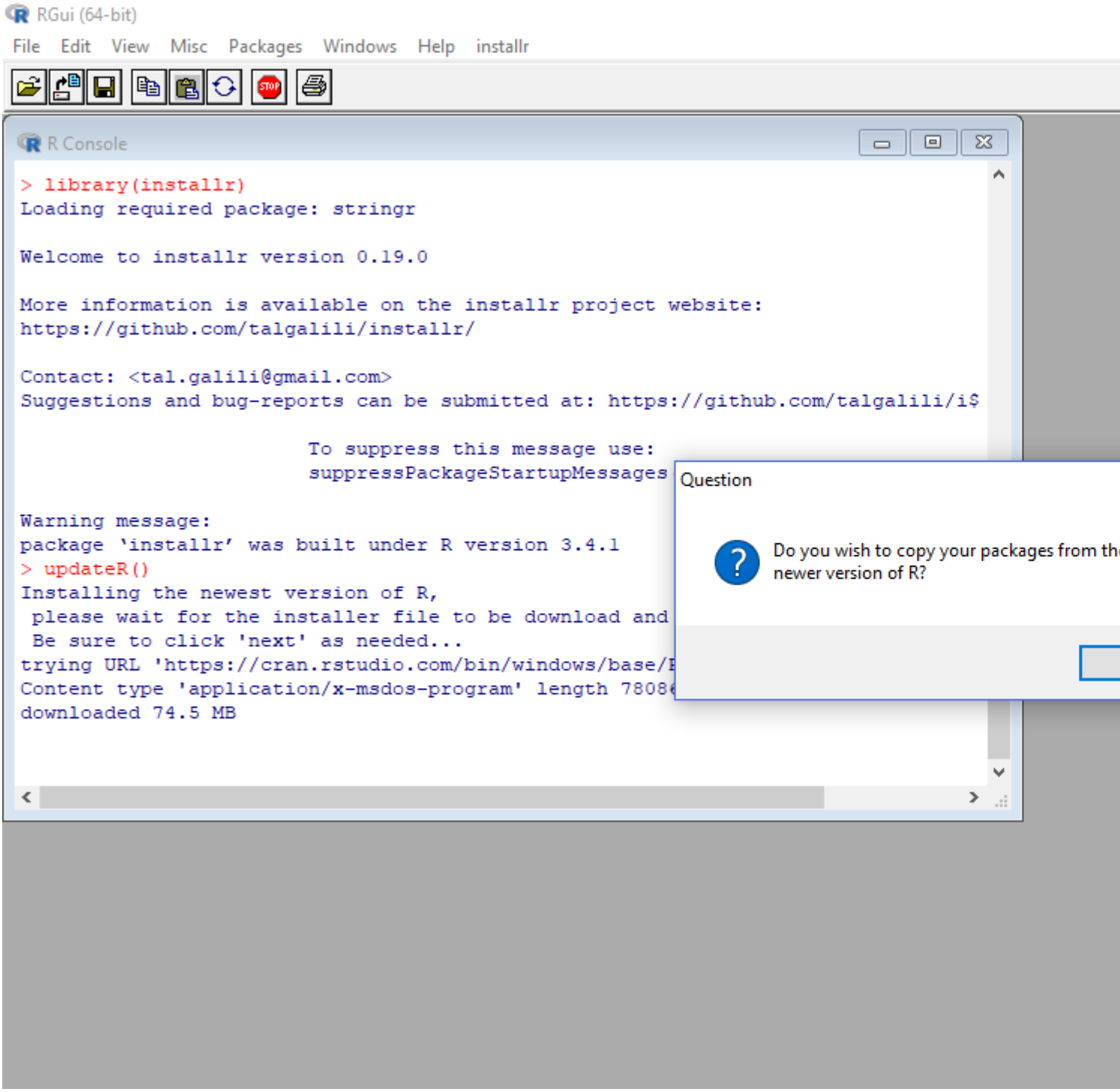

www.dbooks.org



Deciding on the old packages

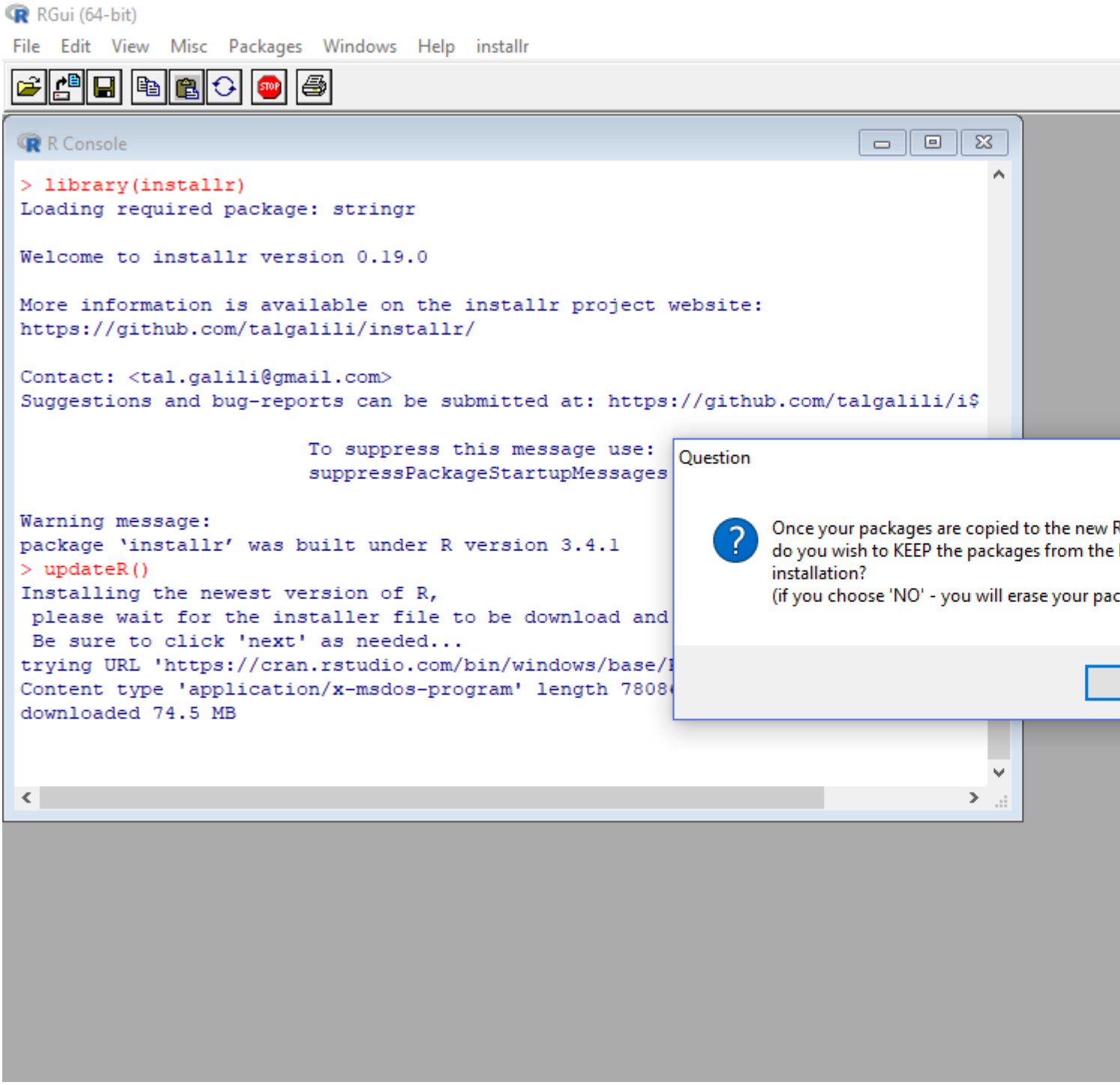
Once the installation is finished click the Finish button.

Now it asks if you want to copy your packages fro the older version of R to Newer version of R. Once you choose yes all the package are copied to the newer version of R.

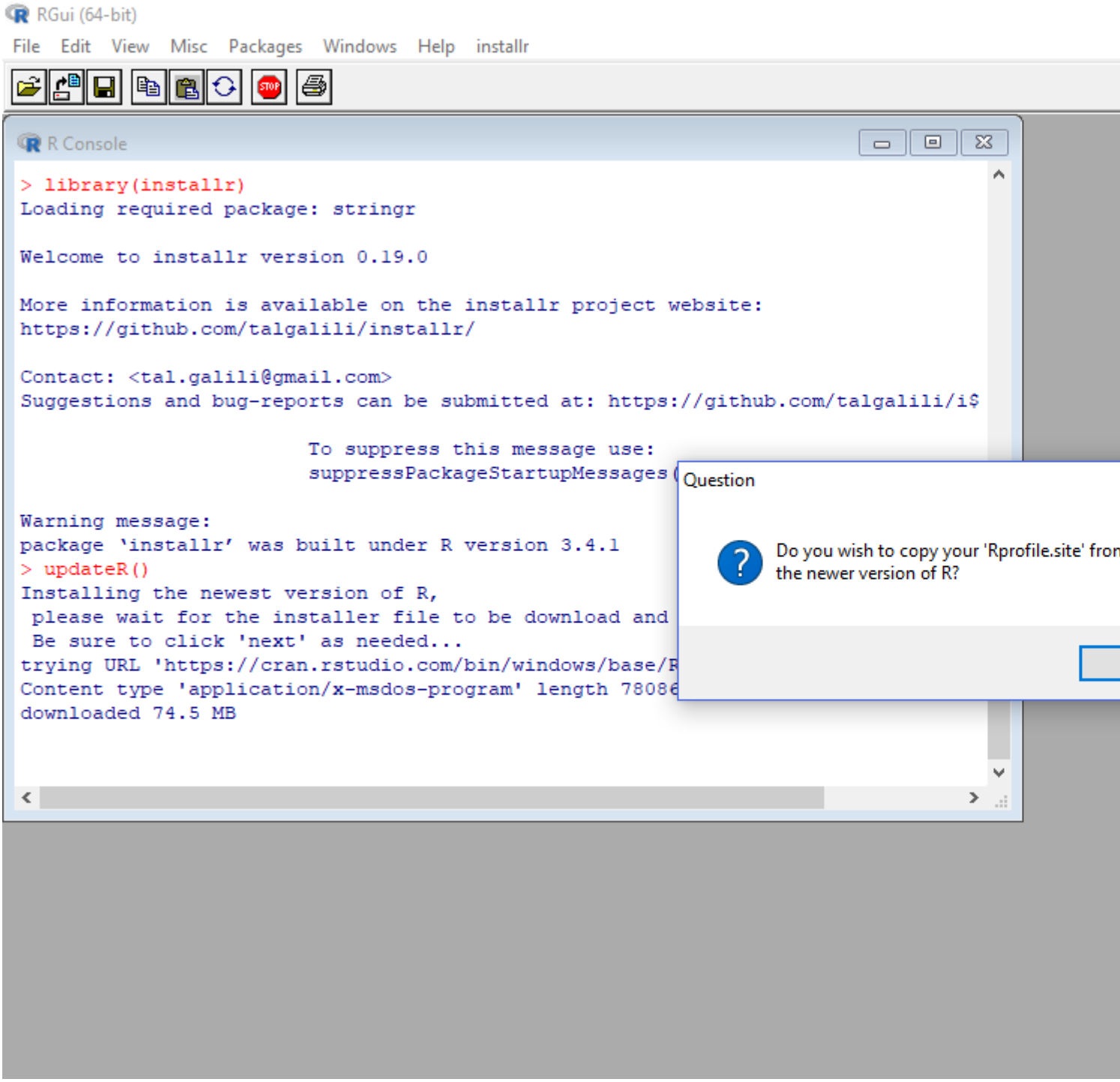


After that you can choose if you still want to keep the old packages or delete.

www.dbooks.org



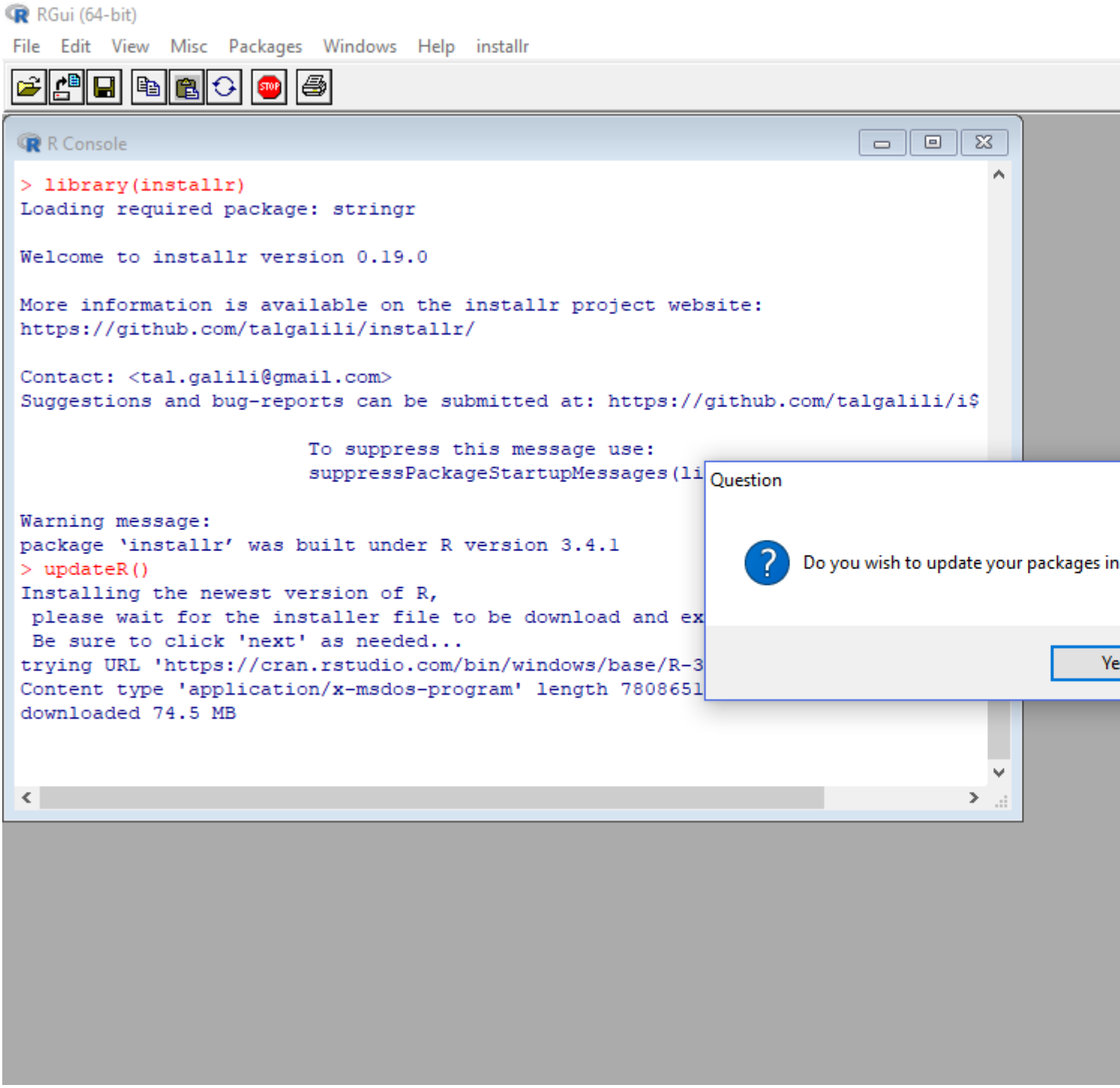
You can even move your Rprofile.site from older version to keep all your customised settings.



Updating Packages

You can update your installed packages once the updating of R is done.

www.dbooks.org



Once its done Restart R and enjoy exploring.

Check R Version

You can check R Version using the console

```
version
```

Read Updating R version online: https://riptutorial.com/r/topic/10729/updating-r-version

Chapter 125: Using pipe assignment in your own package %<>%: How to ?

Introduction

In order to use the pipe in a user-created package, it must be listed in the NAMESPACE like any other function you choose to import.

Examples

Putting the pipe in a utility-functions file

One option for doing this is to export the pipe from within the package itself. This may be done in the 'traditional' `zzz.R` or `utils.R` files that many packages utilise for useful little functions that are not exported as part of the package. For example, putting:

```
#' Pipe operator
#'
#' @name %>%
#' @rdname pipe
#' @keywords internal
#' @export
#' @importFrom magrittr %>%
#' @usage lhs \%>\% rhs
NULL
```

Read Using pipe assignment in your own package %<>%: How to ? online: https://riptutorial.com/r/topic/10547/using-pipe-assignment-in-your-own-package-------how-to--

www.dbooks.org

Chapter 126: Using texreg to export models in a paper-ready way

Introduction

The texreg package helps to export a model (or several models) in a neat paper-ready way. The result may be exported as HTML or .doc (MS Office Word).

Remarks

Links

- CRAN page

Examples

Printing linear regression results

```
# models
fit1 <- lm(mpg ~ wt, data = mtcars)
fit2 <- lm(mpg ~ wt+hp, data = mtcars)
fit3 <- lm(mpg ~ wt+hp+cyl, data = mtcars)

# export to html
texreg::htmlreg(list(fit1,fit2,fit3),file='models.html')


# export to doc
texreg::htmlreg(list(fit1,fit2,fit3),file='models.doc')
```

The result looks like a table in a paper.

	Model 1	Model 2	Model 3
(Intercept)	37.29***	37.23***	38.75***
	(1.88)	(1.60)	(1.79)
wt	-5.34***	-3.88***	-3.17***
	(0.56)	(0.63)	(0.74)
hp		-0.03**	-0.02
		(0.01)	(0.01)
cyl			-0.94
			(0.55)
R2	0.75	0.83	0.84
Adj. R2	0.74	0.81	0.83
Num. obs.	32	32	32
RMSE	3.05	2.59	2.51

***p < 0.001, **p < 0.01, *p < 0.05

Statistical models

There are several additional handy parameters in `texreg::htmlreg()` function. Here is a use case for the most helpful parameters.

```
# export to html
texreg::htmlreg(list(fit1,fit2,fit3),file='models.html',
                single.row = T,
                custom.model.names = LETTERS[1:3],
                leading.zero = F,
                digits = 3)
```

Which result in a table like this

	A	B	C
(Intercept)	37.285 (1.878)***	37.227 (1.599)***	38.752 (1.787)***
wt	-5.344 (0.559)***	-3.878 (0.633)***	-3.167 (0.741)***
hp		-0.032 (0.009)**	-0.018 (0.012)
cyl			-0.942 (0.551)
R2	0.753	0.827	0.843
Adj. R2	0.745	0.815	0.826
Num. obs.	32	32	32
RMSE	3.046	2.593	2.512

***p < 0.001, **p < 0.01, *p < 0.05

Statistical models

Read Using texreg to export models in a paper-ready way online:
https://riptutorial.com/r/topic/9037/using-texreg-to-export-models-in-a-paper-ready-way

www.dbooks.org

Chapter 127: Variables

Examples

Variables, data structures and basic Operations

In R, data objects are manipulated using named data structures. The names of the objects might be called "variables" although that term does not have a specific meaning in the official R documentation. R names are *case sensitive* and may contain alphanumeric characters(`a-z`,`A-z`,`0-9`), the dot/period(`.`) and underscore(_). To create names for the data structures, we have to follow the following rules:

- Names that start with a digit or an underscore (e.g. `1a`), or names that are valid numerical expressions (e.g. `.11`), or names with dashes ('-') or spaces can only be used when they are quoted: `` `1a` `` and `` `.11` ``. The names will be printed with backticks:

```
list( '.11' ="a")
#$`.11`
#[1] "a"
```

- All other combinations of alphanumeric characters, dots and underscores can be used freely, where reference with or without backticks points to the same object.

- Names that begin with `.` are considered system names and are not always visible using the `ls()`-function.

There is no restriction on the number of characters in a variable name.

Some examples of valid object names are: `foobar`, `foo.bar`, `foo_bar`, `.foobar`

In R, variables are assigned values using the infix-assignment operator `<-`. The operator `=` can also be used for assigning values to variables, however its proper use is for associating values with parameter names in function calls. Note that omitting spaces around operators may create confusion for users. The expression `a<-1` is parsed as assignment (`a <- 1`) rather than as a logical comparison (`a < -1`).

```
> foo <- 42
> fooEquals = 43
```

So `foo` is assigned the value of `42`. Typing `foo` within the console will output `42`, while typing `fooEquals` will output `43`.

```
> foo
[1] 42
> fooEquals
[1] 43
```

The following command assigns a value to the variable named `x` and prints the value simultaneously:

```
> (x <- 5)
[1] 5
# actually two function calls: first one to `<-`; second one to the `()`-function
> is.function(`(`)
[1] TRUE  # Often used in R help page examples for its side-effect of printing.
```

It is also possible to make assignments to variables using `->`.

```
> 5 -> x
> x
[1] 5
>
```

Types of data structures

There are no scalar data types in R. Vectors of length-one act like scalars.

- Vectors: Atomic vectors must be sequence of same-class objects.: a sequence of numbers, or a sequence of logicals or a sequence of characters. `v <- c(2, 3, 7, 10)`, `v2 <- c("a", "b", "c")` are both vectors.
- Matrices: A matrix of numbers, logical or characters. `a <- matrix(data = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), nrow = 4, ncol = 3, byrow = F)`. Like vectors, matrix must be made of same-class elements. To extract elements from a matrix rows and columns must be specified: `a[1,2]` returns `[1] 5` that is the element on the first row, second column.
- Lists: concatenation of different elements `mylist <- list (course = 'stat', date = '04/07/2009', num_isc = 7, num_cons = 6, num_mat = as.character(c(45020, 45679, 46789, 43126, 42345, 47568, 45674)), results = c(30, 19, 29, NA, 25, 26 ,27) )`. Extracting elements from a list can be done by name (if the list is named) or by index. In the given example `mylist$results` and `mylist[[6]]` obtains the same element. Warning: if you try `mylist[6]`, R wont give you an error, but it extract the result as a list. While `mylist[[6]][2]` is permitted (it gives you 19), `mylist[6][2]` gives you an error.
- data.frame: object with columns that are vectors of equal length, but (possibly) different types. They are not matrices. `exam <- data.frame(matr = as.character(c(45020, 45679, 46789, 43126, 42345, 47568, 45674)), res_S = c(30, 19, 29, NA, 25, 26, 27), res_O = c(3, 3, 1, NA, 3, 2, NA), res_TOT = c(30,22,30,NA,28,28,27))`. Columns can be read by name `exam$matr`, `exam[, 'matr']` or by index `exam[1]`, `exam[,1]`. Rows can also be read by name `exam['rowname', ]` or index `exam[1,]`. Dataframes are actually just lists with a particular structure (rownames-attribute and equal length components)

Common operations and some cautionary advice

Default operations are done element by element. See `?Syntax` for the rules of operator precedence. Most operators (and may other functions in base R) have recycling rules that allow arguments of unequal length. Given these objects:

www.dbooks.org

Example objects

```
> a <- 1
> b <- 2
> c <- c(2,3,4)
> d <- c(10,10,10)
> e <- c(1,2,3,4)
> f <- 1:6
> W <- cbind(1:4,5:8,9:12)
> Z <- rbind(rep(0,3),1:3,rep(10,3),c(4,7,1))
```

Some vector operations

```
> a+b # scalar + scalar
[1] 3
> c+d # vector + vector
[1] 12 13 14
> a*b # scalar * scalar
[1] 2
> c*d # vector * vector (componentwise!)
[1] 20 30 40
> c+a # vector + scalar
[1] 3 4 5
> c^2 #
[1]  4  9 16
> exp(c)
[1]  7.389056 20.085537 54.598150
```

Some vector operation Warnings!

```
> c+e # warning but.. no errors, since recycling is assumed to be desired.
[1] 3 5 7 6
Warning message:
In c + e : longer object length is not a multiple of shorter object length
```

R sums what it can and then reuses the shorter vector to fill in the blanks... The warning was given only because the two vectors have lengths that are not exactly multiples. c+f # no warning whatsoever.

Some Matrix operations Warning!

```
> Z+W # matrix + matrix #(componentwise)
> Z*W # matrix* matrix#(Standard product is always componentwise)
```

To use a matrix multiply: V %*% W

```
> W + a # matrix+ scalar is still componentwise
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
[4,]    5    9   13

> W + c # matrix + vector... : no warnings and R does the operation in a column-wise manner
     [,1] [,2] [,3]
[1,]    3    8   13
[2,]    5   10   12
[3,]    7    9   14
[4,]    6   11   16
```

"Private" variables

A leading dot in a name of a variable or function in R is commonly used to denote that the variable or function is meant to be hidden.

So, declaring the following variables

```
> foo <- 'foo'
> .foo <- 'bar'
```

And then using the `ls` function to list objects will only show the first object.

```
> ls()
[1] "foo"
```

However, passing `all.names = TRUE` to the function will show the 'private' variable

```
> ls(all.names = TRUE)
[1] ".foo"          "foo"
```

Read Variables online: https://riptutorial.com/r/topic/9013/variables

www.dbooks.org

Chapter 128: Web Crawling in R

Examples

Standard scraping approach using the RCurl package

We try to extract imdb top chart movies and ratings

```
R> library(RCurl)
R> library(XML)
R> url <- "http://www.imdb.com/chart/top"
R> top <- getURL(url)
R> parsed_top <- htmlParse(top, encoding = "UTF-8")
R> top_table <- readHTMLTable(parsed_top)[[1]]
R> head(top_table[1:10, 1:3])


Rank & Title IMDb Rating
1 1. The Shawshank Redemption (1994) 9.2
2 2. The Godfather (1972) 9.2
3 3. The Godfather: Part II (1974) 9.0
4 4. The Dark Knight (2008) 8.9
5 5. Pulp Fiction (1994) 8.9
6 6. The Good, the Bad and the Ugly (1966) 8.9
7 7. Schindler's List (1993) 8.9
8 8. 12 Angry Men (1957) 8.9
9 9. The Lord of the Rings: The Return of the King (2003) 8.9
10 10. Fight Club (1999) 8.8
```

Read Web Crawling in R online: https://riptutorial.com/r/topic/4336/web-crawling-in-r

Chapter 129: Web scraping and parsing

Remarks

Scraping refers to using a computer to retrieve the code of a webpage. Once the code is obtained, it must be *parsed* into a useful form for further use in R.

Base R does not have many of the tools required for these processes, so scraping and parsing are typically done with packages. Some packages are most useful for scraping (`RSelenium`, `httr`, `curl`, `RCurl`), some for parsing (`XML`, `xml2`), and some for both (`rvest`).

A related process is scraping a web API, which unlike a webpage returns data intended to be machine-readable. Many of the same packages are used for both.

Legality

Some websites object to being scraped, whether due to increased server loads or concerns about data ownership. If a website forbids scraping in it Terms of Use, scraping it is illegal.

Examples

Basic scraping with rvest

`rvest` is a package for web scraping and parsing by Hadley Wickham inspired by Python's Beautiful Soup. It leverages Hadley's `xml2` package's `libxml2` bindings for HTML parsing.

As part of the tidyverse, `rvest` is piped. It uses

- `xml2::read_html` to scrape the HTML of a webpage,
- which can then be subset with its `html_node` and `html_nodes` functions using CSS or XPath selectors, and
- parsed to R objects with functions like `html_text` and `html_table`.

To scrape the table of milestones from the Wikipedia page on R, the code would look like

```
library(rvest)

url <- 'https://en.wikipedia.org/wiki/R_(programming_language)'

        # scrape HTML from website
url %>% read_html() %>%
    # select HTML tag with class="wikitable"
    html_node(css = '.wikitable') %>%
    # parse table into data.frame
    html_table() %>%
    # trim for printing
    dplyr::mutate(Description = substr(Description, 1, 70))
```


www.dbooks.org

```
##    Release       Date                                                Description
## 1     0.16            This is the last alpha version developed primarily by Ihaka
## 2     0.49 1997-04-23 This is the oldest source release which is currently availab
## 3     0.60 1997-12-05 R becomes an official part of the GNU Project. The code is h
## 4   0.65.1 1999-10-07 First versions of update.packages and install.packages funct
## 5      1.0 2000-02-29 Considered by its developers stable enough for production us
## 6      1.4 2001-12-19 S4 methods are introduced and the first version for Mac OS X
## 7      2.0 2004-10-04 Introduced lazy loading, which enables fast loading of data
## 8      2.1 2005-04-18 Support for UTF-8 encoding, and the beginnings of internatio
## 9     2.11 2010-04-22                          Support for Windows 64 bit systems.
## 10    2.13 2011-04-14 Adding a new compiler function that allows speeding up funct
## 11    2.14 2011-10-31 Added mandatory namespaces for packages. Added a new paralle
## 12    2.15 2012-03-30 New load balancing functions. Improved serialization speed f
## 13     3.0 2013-04-03 Support for numeric index values 231 and larger on 64 bit sy
```

While this returns a data.frame, note that as is typical for scraped data, there is still further data cleaning to be done: here, formatting dates, inserting `NA`s, and so on.

Note that data in a less consistently rectangular format may take looping or other further munging to successfully parse. If the website makes use of jQuery or other means to insert content, `read_html` may be insufficient to scrape, and a more robust scraper like `RSelenium` may be necessary.

Using rvest when login is required

I common problem encounter when scrapping a web is how to enter a userid and password to log into a web site.

In this example which I created to track my answers posted here to stack overflow. The overall flow is to login, go to a web page collect information, add it a dataframe and then move to the next page.

```
library(rvest)

#Address of the login webpage
login<-
"https://stackoverflow.com/users/login?ssrc=head&returnurl=http%3a%2f%2fstackoverflow.com%2f"

#create a web session with the desired login address
pgsession<-html_session(login)
pgform<-html_form(pgsession)[[2]]  #in this case the submit is the 2nd form
filled_form<-set_values(pgform, email="*****", password="*****")
submit_form(pgsession, filled_form)

#pre allocate the final results dataframe.
results<-data.frame()

#loop through all of the pages with the desired info
for (i in 1:5)
{
  #base address of the pages to extract information from
  url<-"http://stackoverflow.com/users/**********?tab=answers&sort=activity&page="
  url<-paste0(url, i)
  page<-jump_to(pgsession, url)
```

```
  #collect info on the question votes and question title
  summary<-html_nodes(page, "div .answer-summary")
  question<-matrix(html_text(html_nodes(summary, "div"), trim=TRUE), ncol=2, byrow = TRUE)

  #find date answered, hyperlink and whether it was accepted
  dateans<-html_node(summary, "span") %>% html_attr("title")
  hyperlink<-html_node(summary, "div a") %>% html_attr("href")
  accepted<-html_node(summary, "div") %>% html_attr("class")

  #create temp results then bind to final results
  rtemp<-cbind(question, dateans, accepted, hyperlink)
  results<-rbind(results, rtemp)
}

#Dataframe Clean-up
names(results)<-c("Votes", "Answer", "Date", "Accepted", "HyperLink")
results$Votes<-as.integer(as.character(results$Votes))
results$Accepted<-ifelse(results$Accepted=="answer-votes default", 0, 1)
```

The loop in this case is limited to only 5 pages, this needs to change to fit your application. I replaced the user specific values with ******, hopefully this will provide some guidance for you problem.

Read Web scraping and parsing online: https://riptutorial.com/r/topic/2890/web-scraping-and-parsing

www.dbooks.org

Chapter 130: Writing functions in R

Examples

Named functions

R is full of functions, it is after all a functional programming language, but sometimes the precise function you need isn't provided in the Base resources. You could conceivably install a package containing the function, but maybe your requirements are just so specific that no pre-made function fits the bill? Then you're left with the option of making your own.

A function can be very simple, to the point of being being pretty much pointless. It doesn't even need to take an argument:

```
one <- function() { 1 }
one()
[1] 1

two <- function() { 1 + 1 }
two()
[1] 2
```

What's between the curly braces `{ }` is the function proper. As long as you can fit everything on a single line they aren't strictly needed, but can be useful to keep things organized.

A function can be very simple, yet highly specific. This function takes as input a vector (`vec` in this example) and outputs the same vector with the vector's length (6 in this case) subtracted from each of the vector's elements.

```
vec <- 4:9
subtract.length <- function(x) { x - length(x) }
subtract.length(vec)
[1] -2 -1  0  1  2  3
```

Notice that `length()` is in itself a pre-supplied (i.e. *Base*) function. You can of course use a previously self-made function within another self-made function, as well as assign variables and perform other operations while spanning several lines:

```
vec2 <- (4:7)/2

msdf <- function(x, multiplier=4) {
    mult <- x * multiplier
    subl <- subtract.length(x)
    data.frame(mult, subl)
}

msdf(vec2, 5)
  mult subl
1 10.0 -2.0
2 12.5 -1.5
```

```
3 15.0 -1.0
4 17.5 -0.5
```

`multiplier=4` makes sure that `4` is the default value of the argument `multiplier`, if no value is given when calling the function `4` is what will be used.

The above are all examples of *named* functions, so called simply because they have been given names (`one`, `two`, `subtract.length` etc.)

Anonymous functions

An anonymous function is, as the name implies, not assigned a name. This can be useful when the function is a part of a larger operation, but in itself does not take much place. One frequent use-case for anonymous functions is within the `*apply` family of Base functions.

Calculate the root mean square for each column in a `data.frame`:

```
df <- data.frame(first=5:9, second=(0:4)^2, third=-1:3)

apply(df, 2, function(x) { sqrt(sum(x^2)) })
    first    second     third
15.968719 18.814888  3.872983
```

Create a sequence of step-length one from the smallest to the largest value for each row in a matrix.

```
x <- sample(1:6, 12, replace=TRUE)
mat <- matrix(x, nrow=3)

apply(mat, 1, function(x) { seq(min(x), max(x)) })
```

An anonymous function can also stand on its own:

```
(function() { 1 })()
[1] 1
```

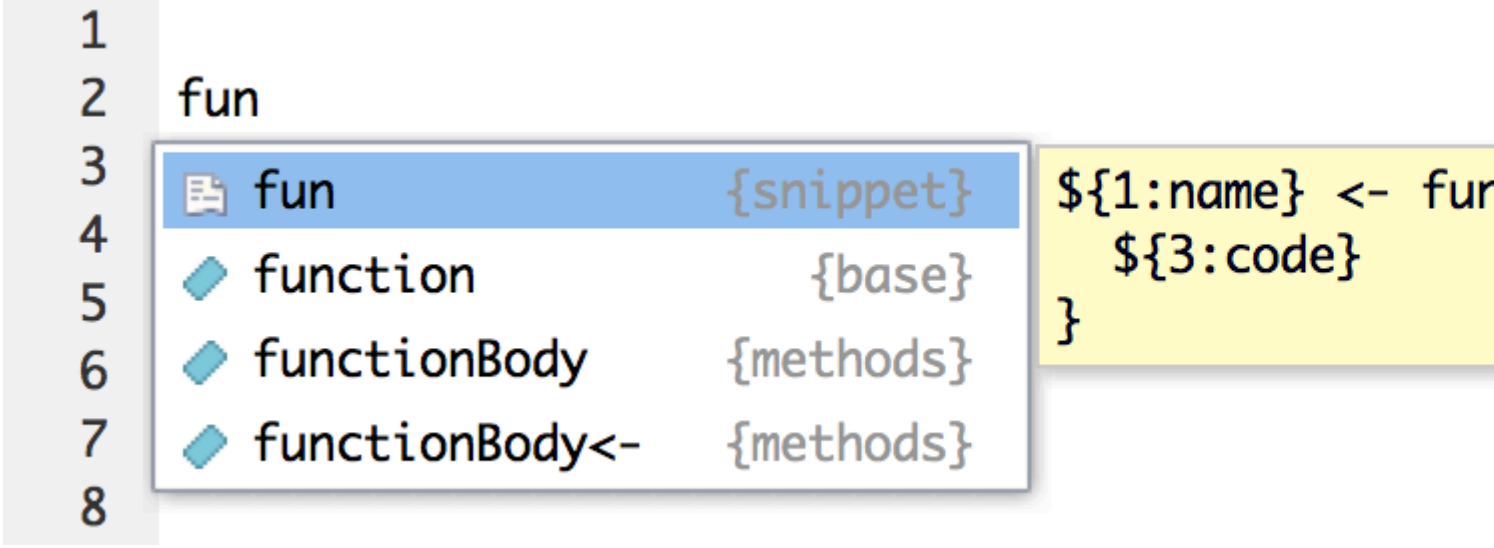
is equivalent to

```
f <- function() { 1 })
f()
[1] 1
```

RStudio code snippets

This is just a small hack for those who use self-defined functions often.
Type "fun" RStudio IDE and hit TAB.

www.dbooks.org



The result will be a skeleton of a new function.

```
name <- function(variables) {

}
```

One can easily define their own snippet template, i.e. like the one below

```
name <- function(df, x, y) {
        require(tidyverse)
        out <-
        return(out)
}
```

The option is `Edit Snippets` in the `Global Options -> Code` menu.

Passing column names as argument of a function

Sometimes one would like to pass names of columns from a data frame to a function. They may be provided as strings and used in a function using `[[`. Let's take a look at the following example, which prints to R console basic stats of selected variables:

```
basic.stats <- function(dset, vars){
    for(i in 1:length(vars)){
        print(vars[i])
        print(summary(dset[[vars[i]]]))
    }
}

basic.stats(iris, c("Sepal.Length", "Petal.Width"))
```

As a result of running above given code, names of selected variables and their basic summary statistics (minima, first quantiles, medians, means, third quantiles and maxima) are printed in R console. The code `dset[[vars[i]]]` selects i-th element from the argument `vars` and selects a corresponding column in declared input data set `dset`. For example, declaring

`iris[["Sepal.Length"]]` alone would print the `Sepal.Length` column from the `iris` data set as a vector.

Read Writing functions in R online: https://riptutorial.com/r/topic/7937/writing-functions-in-r

www.dbooks.org

Chapter 131: xgboost

Examples

Cross Validation and Tuning with xgboost

```
library(caret) # for dummyVars
library(RCurl) # download https data
library(Metrics) # calculate errors
library(xgboost) # model

################################################################################
# Load data from UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/datasets.html)
urlfile <- 'https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
adults <- read.csv(textConnection(x), header=F)

# adults <-read.csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/adult/adult.data', header=F)
names(adults)=c('age','workclass','fnlwgt','education','educationNum',
                'maritalStatus','occupation','relationship','race',
                'sex','capitalGain','capitalLoss','hoursWeek',
                'nativeCountry','income')
# clean up data
adults$income <- ifelse(adults$income==' <=50K',0,1)
# binarize all factors
library(caret)
dmy <- dummyVars(" ~ .", data = adults)
adultsTrsf <- data.frame(predict(dmy, newdata = adults))
################################################################################

# what we're trying to predict adults that make more than 50k
outcomeName <- c('income')
# list of features
predictors <- names(adultsTrsf)[!names(adultsTrsf) %in% outcomeName]

# play around with settings of xgboost - eXtreme Gradient Boosting (Tree) library
# https://github.com/tqchen/xgboost/wiki/Parameters
# max.depth - maximum depth of the tree
# nrounds - the max number of iterations

# take first 10% of the data only!
trainPortion <- floor(nrow(adultsTrsf)*0.1)

trainSet <- adultsTrsf[ 1:floor(trainPortion/2),]
testSet <- adultsTrsf[(floor(trainPortion/2)+1):trainPortion,]

smallestError <- 100
for (depth in seq(1,10,1)) {
        for (rounds in seq(1,20,1)) {

                # train
                bst <- xgboost(data = as.matrix(trainSet[,predictors]),
                               label = trainSet[,outcomeName],
                               max.depth=depth, nround=rounds,
                               objective = "reg:linear", verbose=0)
                gc()
```

```
                # predict
                predictions <- predict(bst, as.matrix(testSet[,predictors]),
outputmargin=TRUE)
                err <- rmse(as.numeric(testSet[,outcomeName]), as.numeric(predictions))

                if (err < smallestError) {
                        smallestError = err
                        print(paste(depth,rounds,err))
                }
        }
}

cv <- 30
trainSet <- adultsTrsf[1:trainPortion,]
cvDivider <- floor(nrow(trainSet) / (cv+1))

smallestError <- 100
for (depth in seq(1,10,1)) {
        for (rounds in seq(1,20,1)) {
                totalError <- c()
                indexCount <- 1
                for (cv in seq(1:cv)) {
                        # assign chunk to data test
                        dataTestIndex <- c((cv * cvDivider):(cv * cvDivider + cvDivider))
                        dataTest <- trainSet[dataTestIndex,]
                        # everything else to train
                        dataTrain <- trainSet[-dataTestIndex,]

                        bst <- xgboost(data = as.matrix(dataTrain[,predictors]),
                                        label = dataTrain[,outcomeName],
                                        max.depth=depth, nround=rounds,
                                        objective = "reg:linear", verbose=0)
                        gc()
                        predictions <- predict(bst, as.matrix(dataTest[,predictors]),
outputmargin=TRUE)

                        err <- rmse(as.numeric(dataTest[,outcomeName]),
as.numeric(predictions))
                        totalError <- c(totalError, err)
                }
                if (mean(totalError) < smallestError) {
                        smallestError = mean(totalError)
                        print(paste(depth,rounds,smallestError))
                }
        }
}

################################################################################
# Test both models out on full data set

trainSet <- adultsTrsf[ 1:trainPortion,]

# assign everything else to test
testSet <- adultsTrsf[(trainPortion+1):nrow(adultsTrsf),]

bst <- xgboost(data = as.matrix(trainSet[,predictors]),
                label = trainSet[,outcomeName],
                max.depth=4, nround=19, objective = "reg:linear", verbose=0)
pred <- predict(bst, as.matrix(testSet[,predictors]), outputmargin=TRUE)
rmse(as.numeric(testSet[,outcomeName]), as.numeric(pred))
```


www.dbooks.org

```
bst <- xgboost(data = as.matrix(trainSet[,predictors]),
               label = trainSet[,outcomeName],
               max.depth=3, nround=20, objective = "reg:linear", verbose=0)
pred <- predict(bst, as.matrix(testSet[,predictors]), outputmargin=TRUE)
rmse(as.numeric(testSet[,outcomeName]), as.numeric(pred))
```

Read xgboost online: https://riptutorial.com/r/topic/3239/xgboost

Credits

S. No	Chapters	Contributors
1	Getting started with R Language	42-, akraf, Ale, Andrea Cirillo, Andrew Brēza, Axeman, Community, Craig Vermeer, d.b, dotancohen, Francesco Dondi, Frank, G5W, George Bonebright, GForce, Giorgos K, Gregor, H. Pauwelyn, kartoffelsalat, kdopen, Konrad Rudolph, L.V.Rao, lmckeogh, Lovy, Matt, mnoronha, pitosalas, polka, Rahul Saini, RetractedAndRetired, russellpierce, Steve_Corrin, theArun, Thomas, torina, user2100721, while
2	*apply family of functions (functionals)	Benjamin, FisherDisinformation, Gavin Simpson, jcb, Karolis Koncevičius, kneijenhuijs, Maximilian Kohl, nrussell, omar, Robert, seasmith, zacdav
3	.Rprofile	42-, Dirk Eddelbuettel, ikashnitsky, Karolis Koncevičius, Nikos Alexandris, Stedy, Thomas
4	Aggregating data frames	Florian, Frank
5	Analyze tweets with R	Umberto
6	ANOVA	Ben Bolker, DataTx, kneijenhuijs
7	Arima Models	Andrew Bryk, Steve_Corrin
8	Arithmetic Operators	Batanichek, FisherDisinformation, Matt Sandgren, Robert, russellpierce, Tensibai
9	Bar Chart	L.V.Rao
10	Base Plotting	42-, Alexey Shiklomanov, catastrophic-failure, FisherDisinformation, Frank, Giorgos K, K.Daisey, maRtin, MichaelChirico, RamenChef, Robert, symbolrush
11	Bibliography in RMD	J_F, RamenChef
12	boxplot	Carlos Cinelli, Christophe D., Karolis Koncevičius, L.V.Rao
13	caret	highBandWidth, Steve_Corrin
14	Classes	42-, AkselA, David Heckmann, dayne, Frank, Gregor, Jaap, kneijenhuijs, L.V.Rao, Nathan Werth, Steve_Corrin

www.dbooks.org

15	Cleaning data	Derek Corcoran
16	Code profiling	Ben Bolker, Glen Moutrie, Jav, SymbolixAU, USER_1
17	Coercion	d.b
18	Color schemes for graphics	ikashnitsky, munirbe
19	Column wise operation	akrun
20	Combinatorics	Frank, Karolis Koncevičius
21	Control flow structures	Benjamin, David Arenburg, nrussell, Robert, Steve_Corrin
22	Creating packages with devtools	Frank, Lovy
23	Creating reports with RMarkdown	ikashnitsky, Karolis Koncevičius, Martin Schmelzer
24	Creating vectors	alistaire, bartektartanus, Jaap, Karsten W., lmo, Rich Scriven, Robert, Robin Gertenbach, smci, takje
25	Data acquisition	ikashnitsky
26	Data frames	Alex, Andrea Ianni , Batanichek, Carlos Cinelli, Christophe D., DataTx, David Arenburg, David Robinson, dayne, Frank, Gregor, Hack-R, kaksat, R. Schifini, scoa, Sumedh, Thomas, Tomás Barcellos, user2100721
27	data.table	akrun, Allen Wang, bartektartanus, cderv, David, David Arenburg, Dean MacGregor, Eric Lecoutre, Frank, Jaap, jogo, L Co, leogama, Mallick Hossain, micstr, Nathan Werth, oshun, Peter Humburg, Sowmya S. Manian, stanekam, Steve_Corrin, Sumedh, Tensibai, user2100721, Uwe
28	Date and Time	AkselA, alistaire, Angelo, coatless, David Leal, Dean MacGregor, Frank, kneijenhuijs, MichaelChirico, scoa, SymbolixAU, takje, theArun, thelatemail
29	Date-time classes (POSIXct and POSIXlt)	AkselA, alistaire, coatless, Frank, MichaelChirico, SymbolixAU, thelatemail
30	Debugging	James Elderfield, russellpierce
31	Distribution	FisherDisinformation, Frank, L.V.Rao, tenCupMaximum

	Functions	
32	dplyr	4444, Alihan Zıhna, ikashnitsky, Robert, skoh, Sumedh, theArun
33	Expression: parse + eval	YCR
34	Extracting and Listing Files in Compressed Archives	catastrophic-failure, Jeff
35	Factors	42-, Benjamin, dash2, Frank, Gavin Simpson, JulioSergio, kneijenhuijs, Nathan Werth, omar, Rich Scriven, Robert, Steve_Corrin
36	Fault-tolerant/resilient code	Rappster
37	Feature Selection in R -- Removing Extraneous Features	Joy
38	Formula	42-, Axeman, Qaswed, Sathish
39	Fourier Series and Transformations	Hack-R
40	Functional programming	Karolis Koncevičius
41	Generalized linear models	Ben Bolker, YCR
42	Get user input	Ashish, DeveauP
43	ggplot2	akraf, Alex, alistaire, Andrea Cirillo, Artem Klevtsov, Axeman, baptiste, blmoore, Boern, gitblame, ikashnitsky, Jaap, jmax, loki , Matt, Mine Cetinkaya-Rundel, Paolo, smci, Steve_Corrin, Sumedh, Taylor Ostberg, theArun, void, YCR, Yun Ching
44	GPU-accelerated computing	cdeterman
45	Hashmaps	nrussell, russellpierce
46	heatmap and heatmap.2	AndreyAkinshin, Nanami

www.dbooks.org

47	Hierarchical clustering with hclust	Frank, G5W, Tal Galili
48	Hierarchical Linear Modeling	Ben Bolker
49	I/O for database tables	Frank, JHowIX, SommerEngineering
50	I/O for foreign tables (Excel, SAS, SPSS, Stata)	42-, Alex, alistaire, Andrea Cirillo, Carlos Cinelli, Charmgoggles, Crops, Frank, Jaap, Jeromy Anglim, kaksat, Ken S., kitman0804, lmo, Miha, Parfait, polka, Thomas
51	I/O for geographic data (shapefiles, etc.)	Alex, Frank, ikashnitsky
52	I/O for raster images	Frank, loki
53	I/O for R's binary format	Frank, ikashnitsky, Mario, russellpierce, zacdav, zx8754
54	Implement State Machine Pattern using S4 Class	David Leal
55	Input and output	Frank
56	Inspecting packages	Frank, Sowmya S. Manian
57	Installing packages	Aaghaz Hussain, akraf, alko989, Andrew Brēza, Artem Klevtsov, Arun Balakrishnan, Christophe D., CL., Frank, gitblame, Hack-R, hongsy, Jaap, kaksat, kneijenhuijs, lmckeogh, loki, Marc Brinkmann, Miha, Peter Humburg, Pragyaditya Das, Raj Padmanabhan, seasmith, SymbolixAU, theArun, user890739, xamgore, zx8754
58	Introduction to Geographical Maps	4444, AkselA, alistaire, beetroot, Carson, Frank, Hack-R, HypnoGenX, Robert, russellpierce, SymbolixAU, symbolrush
59	Introspection	Jason
60	JSON	SymbolixAU
61	Linear Models (Regression)	Amstell, Ben Bolker, Carl, Carlos Cinelli, David Robinson, fortune_p, Frank, highBandWidth, ikashnitsky, jaySf, Robert, russellpierce, thelatemail, USER_1, WAF
62	Lists	Andrea Ianni , BarkleyBG, dayne, Frank, Hack-R, Hairizuan Noorazman, Peter Humburg, RamenChef

63	lubridate	alistaire, Angelo, Frank, gitblame, Hendrik, scoa
64	Machine learning	loki
65	Matrices	dayne, Frank
66	Meta: Documentation Guidelines	Frank, Gregor, Stephen Leppik, Steve_Corrin
67	Missing values	Amit Kohli, Artem Klevtsov, Axeman, Eric Lecoutre, Frank, Gregor, Jaap, kitman0804, lmo, seasmith, Steve_Corrin, theArun, user2100721
68	Modifying strings by substitution	Alex, David Leal, Frank
69	Natural language processing	CptNemo
70	Network analysis with the igraph package	Boysenb3rry
71	Non-standard evaluation and standard evaluation	PAC
72	Numeric classes and storage modes	Frank, Steve_Corrin
73	Object-Oriented Programming in R	Jon Ericson, rcorty
74	Parallel processing	Artem Klevtsov, jameselmore, K.Daisey, lmo, loki, russellpierce
75	Pattern Matching and Replacement	Abdou, Alex, Artem Klevtsov, David Arenburg, David Leal, Frank, Gavin Simpson, Jaap, NWaters, R. Schifini, SommerEngineering, Steve_Corrin, Tensibai, thelatemail, user2100721
76	Performing a Permutation Test	Stephen Leppik, tenCupMaximum
77	Pipe operators (%>% and others)	42-, Alexandru Papiu, Alihan Zıhna, alistaire, AndreyAkinshin, Artem Klevtsov, Atish, Axeman, Benjamin, Carlos Cinelli, CMichael, DrPositron, Franck Dernoncourt, Frank, Gal Dreiman , Gavin Simpson, Gregor, ikashnitsky, James McCalden, Kay Brodersen, Matt, polka, RamenChef, Ryan Hilbert, Sam Firke, seasmith, Shawn Mehan, Simplans, Spacedman, SymbolixAU,

www.dbooks.org

		thelatemail, tomw, TriskalJM, user2100721
78	Pivot and unpivot with data.table	Sun Bee
79	Probability Distributions with R	Pankaj Sharma
80	Publishing	Frank
81	R code vectorization best practices	Axeman, David Arenburg, snaut
82	R in LaTeX with knitr	JHowIX
83	R Markdown Notebooks (from RStudio)	dmail
84	R memento by examples	Lovy
85	Random Forest Algorithm	G5W
86	Random Numbers Generator	bartektartanus, FisherDisinformation, Karolis Koncevičius, Miha , mnoronha
87	Randomization	TARehman
88	Raster and Image Analysis	Frank, loki
89	Rcpp	Artem Klevtsov, coatless, Dirk Eddelbuettel
90	Reading and writing strings	42-, 4444, abhiieor, cdrini, dotancohen, Frank, Gregor, kdopen, Rich Scriven, Thomas, Uwe
91	Reading and writing tabular data in plain-text files (CSV, TSV, etc.)	a.powell, Aaghaz Hussain, abhiieor, Alex, alistaire, Andrea Cirillo, bartektartanus, Carl Witthoft, Carlos Cinelli, catastrophic-failure, cdrini, Charmgoggles, Crops, DaveRGP, David Arenburg, Dawny33, Derwin McGeary, EDi, Eric Lecoutre, FoldedChromatin, Frank, Gavin Simpson, gitblame, Hairizuan Noorazman, herbaman, ikashnitsky, Jaap, Jeromy Anglim, JHowIX, joeyreid, Jordan Kassof, K.Daisey, kitman0804, kneijenhuijs, lmo, loki, Miha, PAC, polka, russellpierce, Sam Firke, stats-hb, Thomas, Uwe, zacdav, zelite, zx8754
92	Recycling	Frank, USER_1

93	Regular Expression Syntax in R	Alexey Shiklomanov
94	Regular Expressions (regex)	42-, Benjamin, David Leal, etienne, Frank, MichaelChirico, PAC
95	Reproducible R	Charmgoggles, Frank, ikashnitsky
96	Reshape using tidyr	Charmgoggles, Frank, Jeromy Anglim, SymbolixAU, user2100721
97	Reshaping data between long and wide forms	Charmgoggles, David Arenburg, demonplus, Frank, Jeromy Anglim, kneijenhuijs, lmo, Steve_Corrin, SymbolixAU, takje, user2100721, zx8754
98	RESTful R Services	YCR
99	RMarkdown and knitr presentation	Martin Schmelzer, YCR
100	RODBC	akrun, Hack-R, Parfait, Tim Coker
101	roxygen2	DeveauP, PAC
102	Run-length encoding	Frank, josliber, Psidom
103	Scope of variables	Artem Klevtsov, K.Daisey, RamenChef
104	Set operations	DeveauP, FisherDisinformation, Frank
105	Shiny	alistaire, CClaire, Christophe D., JvH, russellpierce, SymbolixAU, tuomastik, zx8754
106	Solving ODEs in R	J_F
107	Spark API (SparkR)	Maximilian Kohl
108	spatial analysis	beetroot, ikashnitsky, loki, maRtin
109	Speeding up tough-to-vectorize code	egnha, josliber
110	Split function	Eric Lecoutre, etienne, josliber, Sathish, Tensibai, thelatemail, user2100721
111	sqldf	Hack-R, Miha
112	Standardize analyses by writing standalone R scripts	akraf, herbaman

www.dbooks.org

113	String manipulation with stringi package	bartektartanus, FisherDisinformation
114	strsplit function	lmo
115	Subsetting	42-, Agriculturist, alexis_laz, alistaire, dayne, Frank, Gavin Simpson, Gregor, L.V.Rao, Mario, mrip, RamenChef, smci, user2100721, zx8754
116	Survival analysis	42-, Axeman, Hack-R, Marcin Kosiński
117	Text mining	Hack-R
118	The character class	Frank, Steve_Corrin
119	The Date class	alistaire, coatless, Frank, L.V.Rao, MichaelChirico, Steve_Corrin
120	The logical class	42-, Frank, Gregor, L.V.Rao, Steve_Corrin
121	tidyverse	David Robinson, egnha, Frank, ikashnitsky, RamenChef, Sumedh
122	Time Series and Forecasting	Andras Deak, Andrew Bryk, coatless, Hack-R, JGreenwell, Pankaj Sharma, Steve_Corrin, µ Muthupandian
123	Updating R and the package library	Eric Lecoutre
124	Updating R version	dmail
125	Using pipe assignment in your own package %<>%: How to ?	RobertMc
126	Using texreg to export models in a paper-ready way	Frank, ikashnitsky
127	Variables	42-, Ale, Axeman, Craig Vermeer, Frank, L.V.Rao, lmckeogh
128	Web Crawling in R	Pankaj Sharma
129	Web scraping and parsing	alistaire, Dave2e
130	Writing functions in R	AkselA, ikashnitsky, kaksat

131	xgboost	Hack-R

www.dbooks.org