The R Environment

FRE6871 & FRE7241, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

January 30, 2018

Internal R Help and Documentation

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

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

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

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

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

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

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

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

R Online Help and Documentation

R Cheat Sheets

Cheat Sheets are a fast way to find what you want

https://www.rstudio.com/resources/cheatsheets/

R Programming Wikibook

Wikibooks are crowdsourced textbooks

http://en.wikibooks.org/wiki/R_Programming/

R FAQ

Frequently Asked Questions about R

http://cran.r-project.org/doc/FAQ/R-FAQ.html

R-seek Online Search Tool

R-seek allows online searches specific to the R language

http://www.rseek.org/

R-help Mailing List

R-help is a very comprehensive Q&A mailing list

https://stat.ethz.ch/mailman/listinfo/r-help

R Style Guides

DataCamp R style guide

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

Google R style guide

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

Stack Exchange

Stack Overflow

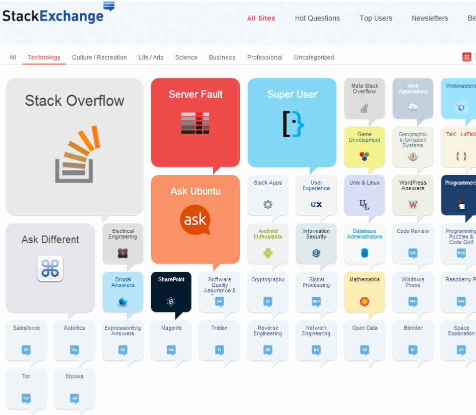
Stack Overflow is a Q&A forum for computer programming, and is part of Stack Exchange

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

Stack Exchange

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

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



RStudio Support

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

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

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

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

R Online Books and Courses

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

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

Endmemo web book

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

Quick-R by Robert Kabacoff

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

R for Beginners by Emmanuel Paradis

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

Cookbook for R by Winston Chang from *RStudio*

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

R Online Interactive Courses

Datacamp Interactive Courses

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

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

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

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

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

Try R

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

R Blogs and Experts

R-Bloggers

R-Bloggers is an aggregator of blogs dedicated to R

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

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

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

Dirk Eddelbuettel

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

http://dirk.eddelbuettel.com/
http://dirk.eddelbuettel.com/code/
http://dirk.eddelbuettel.com/blog/
http://www.rinfinance.com/

Romain Frangois

Romain is an R Enthusiast and Rcpp Hero

http://romainfrancois.blog.free.fr/
http://romainfrancois.blog.free.fr/index.php?tag/graphgallery
http://blog.r-enthusiasts.com/

More R Blogs and Experts

Revolution Analytics Blog

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

RStudio Blog

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

GitHub for Hosting Software Projects Online

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

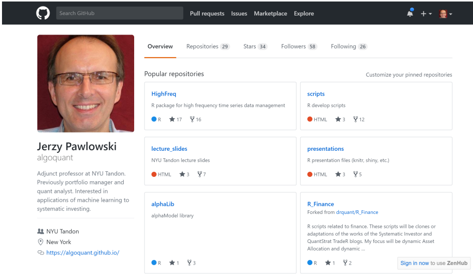
GitHub provides version control using *git* (designed by Linus Torvalds),

Most R projects are now hosted on *GitHub*,

Google uses *GitHub* to host its *tensorflow* library for machine learning:
https://github.com/tensorflow/tensorflow

All the *FRE-7241* and *FRE-6871* lectures are hosted on *GitHub*:
https://github.com/algoquant/lecture_slides
https://github.com/algoquant

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



Overview | Repositories 29 | Stars 34 | Followers 58 | Following 26

Jerzy Pawlowski
algoquant

Adjunct professor at NYU Tandon. Previously portfolio manager and quant analyst. Interested in applications of machine learning to systematic investing.

NYU Tandon
New York
https://algoquant.github.io/

Popular repositories

Customize your pinned repositories

HighFreq
R package for high frequency time series data management
● R ★ 17 16

scripts
R develop scripts
● HTML ★ 3 12

lecture_slides
NYU Tandon lecture slides
● HTML ★ 3 7

presentations
R presentation files (knitr, shiny, etc.)
● HTML ★ 3 5

alphaLib
alphaModel library
● R ★ 1 3

R_Finance
Forked from drquant/R_Finance
R scripts related to finance. These scripts will be clones or adaptations of the works of the Systematic Investor and QuantStrat TradeR blogs. My focus will be dynamic Asset Allocation and dynamic ...
● R ★ 1 2

Sign in now to use ZenHub

What is R?

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

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

Why is R More Difficult Than Other Languages?

R is more difficult than other languages because:

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

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

The R License

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

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

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

http://creativecommons.org

Installing R and *RStudio*

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

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

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

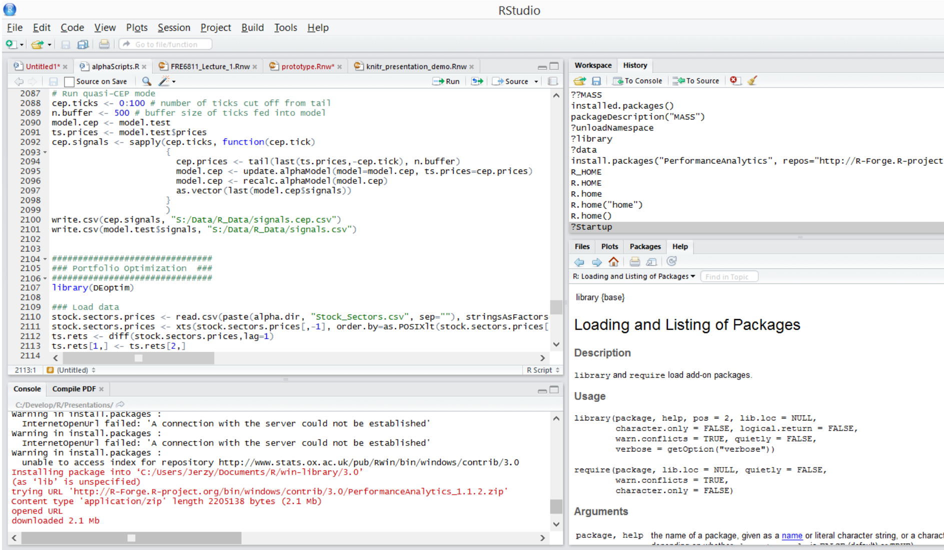
http://cran.r-project.org/

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

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

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

Using *RStudio*



```
# Run quasi-CEP mode
cep.ticks <- 0:100 # number of ticks cut off from tail
n.buffer <- 500 # buffer size of ticks fed into model
model.cep <- model.test
ts.prices <- model.test$prices
cep.signals <- sapply(cep.ticks, function(cep.tick)
                      {
                        cep.prices <- tail(last(ts.prices,-cep.tick), n.buffer)
                        model.cep <- update.alphaModel(model=model.cep, ts.prices=cep.prices)
                        model.cep <- recalc.alphaModel(model.cep)
                        as.vector(last(model.cep$signals))
                      }
                      )
write.csv(cep.signals, "S:/Data/R_Data/signals.cep.csv")
write.csv(model.test$signals, "S:/Data/R_Data/signals.csv")


################################
### Portfolio Optimization  ###
################################
library(DEoptim)

### Load data
stock.sectors.prices <- read.csv(paste(alpha.dir, "Stock_Sectors.csv", sep=""), stringsAsFactors
stock.sectors.prices <- xts(stock.sectors.prices[,-1], order.by=as.POSIXlt(stock.sectors.prices[
ts.rets <- diff(stock.sectors.prices,lag=1)
ts.rets[1,] <- ts.rets[2,]
```

```
Warning in install.packages :
  InternetOpenUrl failed: 'A connection with the server could not be established'
Warning in install.packages :
  InternetOpenUrl failed: 'A connection with the server could not be established'
Warning in install.packages :
  unable to access index for repository http://www.stats.ox.ac.uk/pub/RWin/bin/windows/contrib/3.0
Installing package into ‘C:/Users/Jerzy/Documents/R/win-library/3.0’
(as ‘lib’ is unspecified)
trying URL 'http://R-Forge.R-project.org/bin/windows/contrib/3.0/PerformanceAnalytics_1.1.2.zip'
Content type 'application/zip' length 2205138 bytes (2.1 Mb)
opened URL
downloaded 2.1 Mb
```

```
??MASS
installed.packages()
packageDescription("MASS")
?unloadNamespace
?library
?data
install.packages("PerformanceAnalytics", repos="http://R-Forge.R-project
R_HOME
R.HOME
R.home
R.home("home")
R.home()
?Startup
```

library {base}

Loading and Listing of Packages

Description

library and require load add-on packages.

Usage

```
library(package, help, pos = 2, lib.loc = NULL,
        character.only = FALSE, logical.return = FALSE,
        warn.conflicts = TRUE, quietly = FALSE,
        verbose = getOption("verbose"))

require(package, lib.loc = NULL, quietly = FALSE,
        warn.conflicts = TRUE,
        character.only = FALSE)
```

Arguments

package, help   the name of a package, given as a name or literal character string, or a charact

A First R Session

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

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

R interprets text in quotes ("") as character strings,

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

Typing a symbol or expression evaluates it,

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

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

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

Exploring an R Session

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

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

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

```
> getwd()  # get cwd
> setwd("C:/Develop/R")  # set cwd
> getwd()  # get cwd
```

Get system date and time

Just the date

```
> Sys.time()  # get date and time
[1] "2018-01-30 16:54:46 EST"
>
> Sys.Date()  # get date only
[1] "2018-01-30"
```

The R Workspace

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

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

The function rm() removes objects from the R workspace,

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

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

The function save() saves just the selected objects,

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

```
> var1 <- 3  # define new object
> ls()  # list all objects in workspace
> # list objects starting with "v"
> ls(pattern=glob2rx("v*"))
> # remove all objects starting with "v"
> rm(list=ls(pattern=glob2rx("v*")))
> save.image()  # save workspace to file .RData
> rm(var1)  # remove object
> ls()  # list objects
> load(".RData")
> ls()  # list objects
> var2 <- 5  # define another object
> save(var1, var2,  # save selected objects
+      file="C:/Develop/R/lecture_slides/data/my
> rm(list=ls())  # remove all objects
> ls()  # list objects
> load_ed <- load(file="C:/Develop/R/lecture_sli
> load_ed
> ls()  # list objects
```

The R Workspace (cont.)

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

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

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

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

The function history() displays recent commands,

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

```
> history(5)  # display last 5 commands
> savehistory(file="myfile")  # default is ".Rhi
> loadhistory(file="myfile")  # default is ".Rhi
```

R Session Info

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

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

```
> sessionInfo()  # get R version and other sessi
R version 3.4.2 (2017-09-28)
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 16299)

Matrix products: default

locale:
[1] LC_COLLATE=English_United States.1252
[2] LC_CTYPE=English_United States.1252
[3] LC_MONETARY=English_United States.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United States.1252

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

other attached packages:
[1] knitr_1.16

loaded via a namespace (and not attached):
[1] compiler_3.4.2  magrittr_1.5    tools_3.4.2
[4] stringi_1.1.5   highr_0.6       stringr_1.2.
[7] evaluate_0.10.1
```

Environment Variables

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

For example the environment variables:

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

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

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

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

```
> Sys.getenv()[5:7]  # list some environment var
>
> Sys.getenv("Home")  # get R user HOME director
>
> Sys.setenv(Home="C:/Develop/data")  # set HOME
>
> Sys.getenv("Home")  # get user HOME directory
>
> Sys.getenv("R_home")  # get R_HOME directory
>
> R.home()  # get R_HOME directory
>
> R.home("etc")  # get "etc" sub-directory of R_
```

Global *Options* Settings

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

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

`options("globop")` displays the current value of option `"globop"`,

`getOption("globop")` displays the current value of option `"globop"`,

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

```
> # ?options  # long list of global options
> # interpret strings as characters, not factors
> getOption("stringsAsFactors")  # display optio
> options("stringsAsFactors")  # display option
> options(stringsAsFactors=FALSE)  # set option
> # number of digits printed for numeric values
> options(digits=3)
> # control exponential scientific notation of p
> # positive "scipen" values bias towards fixed
> # negative "scipen" values bias towards scient
> options(scipen=100)
> # maximum number of items printed to console
> options(max.print=30)
> # warning levels options
> # negative - warnings are ignored
> options(warn=-1)
> # zero - warnings are stored and printed after
> options(warn=0)
> # one - warnings are printed as they occur
> options(warn=1)
> # two or larger - warnings are turned into err
> options(warn=2)
> # save all options in variable
> op_tions <- options()
> # restore all options from variable
> options(op_tions)
```

Constructing File Paths

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

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

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

```
> # R startup (site) directory
> paste(R.home(), "etc", sep="/")
[1] "C:/PROGRA~1/R/R-34~1.2/etc"
>
> file.path(R.home(), "etc")  # better way
[1] "C:/PROGRA~1/R/R-34~1.2/etc"
>
> # perform tilde-expansions and convert to read
> normalizePath(file.path(R.home(), "etc"), wins
[1] "C:/Program Files/R/R-3.4.2/etc"
>
> normalizePath(R.home("etc"), winslash="/")
[1] "C:/Program Files/R/R-3.4.2/etc"
```

R System Directories under *Windows*

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

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

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

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

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

File and Directory Listing Functions

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

list.dirs() lists the directories in a given directory,

Sys.glob() lists files matching names obtained from wildcard expansion,

```
> setwd("C:/Develop/R/lecture_slides/data")
> sample(dir(), 5)  # get 5 file names - dir() l
> sample(dir(pattern="csv"), 5)  # list files co
> sample(list.files(R.home()), 5)  # all files i
> sample(list.files(R.home("etc")), 5)  # all fi
> sample(list.dirs(), 5)  # directories in cwd
> list.dirs(R.home("etc"))  # directories in "et
> sample(Sys.glob("*.csv"), 5)
> Sys.glob(R.home("etc"))
```

Invoking an R Session in *Windows*

An R session can run in several different ways:

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

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

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

```
> getwd()  # get cwd
[1] "C:/Develop/R/lecture_slides"
```

R Session Startup

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

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

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

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

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

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

Customizing the R Environment

users can customize their R environments and workspace by creating custom startup files in different working directories. The Renviron and Rprofile files can be placed in any directory `Renviron` files defining environment variables, `Rprofile` files containing code executed at R startup, If R is invoked from a terminal, then the directory from which it's invoked will be sourced. At startup R searches for startup files in the `cwd` and R home directory, every directory can have its own special initialization file environment files (containing environment variables to be set), and .Rprofile files containing R scripts (code),
startup files may contain environment variables, option settings, and other R scripts startup profile file of R code C:/Program Files/R/R-3.1.2/
to process for setting environment variables. executes If no `.Rprofile` file is found in the startup directory, then R looks for a `.Rprofile` file in the user's home directory and uses that (if it exists). The function `getwd()` returns a vector of length 1, with the first element containing a

```
> setwd("C:/Develop/R")
>
> scan(file=".Rprofile", what=character(), sep="
```

The Renviron files

At startup R searches for startup files in the cwd and R home directory,
Environment variables can be supplied as ”symbol=value” pairs on the command line.
environment files (containing environment variables to be set), and .Rprofile files containing R scripts (code),
startup files may contain environment variables, option settings, and other R scripts startup profile file of R code C:/Program Files/R/R-3.1.2/
to process for setting environment variables.
executes If no .Rprofile file is found in the startup directory, then R looks for a .Rprofile file in the user's home directory and uses that (if it exists). The function getwd() returns a vector of length 1, with the first element containing a string with the name of the current working directory (cwd), R sources the .Rprofile file in the current working directory or in the user's home directory (in that order) every directory can have its own custom initialization file

```
> cat("sourcing .Rprofile file\n")
>
>
```

The Rprofile files

At startup R searches for startup files in the cwd and R home directory,
environment files (containing environment variables to be set), and .Rprofile files containing R scripts (code),
startup files may contain environment variables, option settings, and other R scripts startup profile file of R code C:/Program Files/R/R-3.1.2/
to process for setting environment variables. executes If no .Rprofile file is found in the startup directory, then R looks for a .Rprofile file in the user's home directory and uses that (if it exists). R sources the .Rprofile file in the current working directory or in the user's home directory (in that order) every directory can have its own custom initialization file

```
> cat("sourcing .Rprofile file\n")
>
>
```

Environments in R

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

There are three system environments:

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

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

Packages have their own environments,

The enclosure of the base package is the empty environment,

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

The R Search Path

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

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

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

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

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

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

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

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

Extracting Time Series from Environments

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

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

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

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

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

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

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

Referencing Object Components Using with()

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

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

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

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

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

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

Writing Text Strings

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

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

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

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

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

```
> cat("Enter\ttab")  # cat() interprets backslas
> print("Enter\ttab")
>
> my_text <- print("hello")
> my_text  # print() returns its argument
>
> # create string
> my_text <- "Title: My Text\nSome numbers: 1,2,
>
> cat(my_text, file="mytext.txt")  # write to te
>
> cat("Title: My Text",  # write several lines t
+     "Some numbers: 1,2,3,...",
+     "Rprofile files contain code executed at R
+     file="mytext.txt", sep="\n")
>
> save(my_text, file="mytext.RData")  # write to
```

Displaying Numeric Data

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

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

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

Reading Text from Files

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

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

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

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

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

Reading and Writing Data Frames from *Text* Files

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

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

`read.table()` returns a data frame, and coerces non-numeric values to `factors` (unless the `stringsAsFactors=FALSE` option is set),

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

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

```
> # write data frame to text file, and then read
> write.table(data_frame, file="florist.txt")
> data_read <- read.table(file="florist.txt")
> data_read  # a data frame
>
> # write matrix to text file, and then read it
> write.table(mat_rix, file="matrix.txt")
> mat_read <- read.table(file="matrix.txt")
> mat_read  # write.table() coerced matrix to da
> class(mat_read)
> # coerce from data frame back to matrix
> mat_read <- as.matrix(mat_read)
> class(mat_read)
```

Copying Data Frames Between the *clipboard* and R

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

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

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

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

```
> data_frame <- read.table("clipboard", header=T
>
> write.table(x=data_frame, file="clipboard", se
>
> # wrapper function for copying data frame from
> # by default, data is tab delimited, with a he
> read_clip <- function(file="clipboard", sep="\
+                header=TRUE, ...) {
+   read.table(file=file, sep=sep, header=header
+ }  # end read_clip
>
> data_frame <- read_clip()
>
> # wrapper function for copying data frame from
> # by default, data is tab delimited, with a he
> write_clip <- function(data, row.names=FALSE,
+                 col.names=TRUE, ...) {
+   write.table(x=data, file="clipboard", sep="\
+       row.names=row.names, col.names=col.names
+ }  # end write_clip
>
> write_clip(data=data_frame)
>
> # launch spreadsheet-style data editor
> data_frame <- edit(data_frame)
```

Reading and Writing Data Frames From *csv* Files

The easiest way to share data between R and *Excel* is through writing and reading *csv()* files,

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

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

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

`read.csv()` coerces non-numeric values to factors, unless the `stringsAsFactors=FALSE` option is set,

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

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

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

```
> # write data frame to CSV file, and then read
> write.csv(data_frame, file="florist.csv")
> data_read <- read.csv(file="florist.csv",
+                     stringsAsFactors=FALSE)
> data_read  # the row names are read in as extr
> # restore row names
> rownames(data_read) <- data_read[, 1]
> data_read <- data_read[, -1]  # remove extra c
> data_read
> # read data frame, with row names from first c
> data_read <- read.csv(file="florist.csv", row.
> data_read
```

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

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

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

```
> # write data frame to CSV file, without row na
> write.csv(data_frame, row.names=FALSE, file="f
> data_read <- read.csv(file="florist.csv")
> data_read  # a data frame without row names
```

Reading and Writing Matrices From *csv* Files

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

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

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

```
> # write matrix to csv file, and then read it b
> write.csv(mat_rix, file="matrix.csv")
> mat_read <- read.csv(file="matrix.csv", row.na
> mat_read  # read.csv() reads matrix as data fr
> class(mat_read)
> mat_read <- as.matrix(mat_read)  # coerce to m
> identical(mat_rix, mat_read)
> write.csv(mat_rix, row.names=FALSE,
+     file="matrix_ex_rows.csv")
> mat_read <- read.csv(file="matrix_ex_rows.csv"
> mat_read <- as.matrix(mat_read)
> mat_read  # a matrix without row names
```

Reading and Writing Matrices (cont.)

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

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

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

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

Removing row names simplifies the reading and writing of matrices,

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

The function `system.time()` calculates the execution time (in seconds) used to evaluate a given expression,

```
> library(MASS)  # load package "MASS"
> # write to CSV file by row - it's very SLOW!!!
> write.matrix(mat_rix, file="matrix.csv", sep="
> system.time(  # scan reads faster - skip first
+   mat_read <- scan(file="matrix.csv", sep=",",
+               skip=1, what=numeric()))
> col_names <- readLines(con="matrix.csv", n=1)
> col_names  # this is a string!
> col_names <- strsplit(col_names, s=",")[[1]]
> mat_read  # mat_read is a vector, not matrix!
> # coerce by row to matrix
> mat_read <- matrix(mat_read, ncol=length(col_n
+               byrow=TRUE)
> colnames(mat_read) <- col_names  # restore col
> mat_read
```

Reading Matrices Containing Bad Data

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

An example of bad data are character strings in numeric data,

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

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

```
> # read data from a csv file, including row nam
> mat_rix <- read.csv(file="matrix_bad.csv", row
+                    stringsAsFactors=FALSE)
> mat_rix
> class(mat_rix)
> # columns with bad data are character or facto
> sapply(mat_rix, class)
> row_names <- row.names(mat_rix)  # copy row na
> # sapply loop over columns and coerce to numer
> mat_rix <- sapply(mat_rix, as.numeric)
> row.names(mat_rix) <- row_names  # restore row
> # replace NAs with zero
> mat_rix[is.na(mat_rix)] <- 0
> # matrix without NAs
> mat_rix
```

Reading and Writing *zoo* Series From *Text* Files

The package *zoo* contains functions `read.zoo()` and `write.zoo()` for reading and writing *zoo* objects from *text* and *csv* files,

`read.zoo()` and `write.zoo()` are wrappers for `read.table()` and `write.table()`,

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

```
> # create zoo with Date index
> in_dex <- seq(from=as.Date("2013-06-15"),
+             by="day", length.out=100)
> zoo_series <- zoo(cumsum(rnorm(NROW(in_dex))),
+             order.by=in_dex)
> tail(zoo_series, 3)
> # write zoo to text file, and then read it bac
> write.zoo(zoo_series, file="zoo_series.txt")
> zoo_series <- read.zoo("zoo_series.txt")  # re
> tail(zoo_series, 3)
```

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

If the index of a *zoo* series is a *date-time*, then `write.zoo()` writes the date and time fields as separate columns with a *space* between them,

To properly read separate date and time columns from *text* files, `read.zoo()` must be passed arguments `"index.column=list(1,2)"` and `"tz"`,

```
> # create zoo with POSIXct date-time index
> in_dex <- seq(from=as.POSIXct("2013-06-15"),
+               by="hour", length.out=1000)
> zoo_series <- zoo(cumsum(rnorm(length(in_dex))
+               order.by=in_dex)
> tail(zoo_series, 3)
> # write zoo to text file, and then read it bac
> write.zoo(zoo_series, file="zoo_series.txt")
> zoo_series <- read.zoo("zoo_series.txt")  # re
> # time field was read as a separate column
> tail(zoo_series, 3)
> # read and specify that second column is time 
> zoo_series <- read.zoo(file="zoo_series.txt",
+                    index.column=list(1,2),
+                    tz="America/New_York")
> tail(zoo_series, 3)
```

Reading and Writing *zoo* Series From *csv* Files

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

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

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

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

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

```
> # write zoo to CSV file, and then read it back
> write.zoo(zoo_series, file="zoo_series.csv",
+     sep=",", col.names=TRUE)
> zoo_series <- read.zoo(file="zoo_series.csv",
+             header=TRUE, sep=",",
+             drop=FALSE,
+             FUN=as.POSIXct, tz="America/New_Yo
> tail(zoo_series, 3)
> # read zoo from CSV file, with custom date-tim
> zoo_frame <- read.table(file="zoo_series2.csv"
+                   sep=",")
> tail(zoo_frame, 3)  # date-time format mm/dd/y
> zoo_series <- read.zoo(file="zoo_series2.csv",
+             header=TRUE, sep=",",
+             drop=FALSE,
+             FUN=as.POSIXct,
+             tz="America/New_York",
+             format="%m/%d/%Y %H:%M")
> tail(zoo_series, 3)
```

Passing Arguments to the save() Function

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

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

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

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

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

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

Reading and Writing Lists of Objects

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

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

```
> rm(list=ls())  # remove all objects
> # load objects from file
> load_ed <- load(file="my_data.RData")
> load_ed  # vector of loaded objects
> ls()  # list objects
> # assign new values to objects in  global envi
> sapply(load_ed, function(sym_bol) {
+   assign(sym_bol, runif(1), envir=globalenv())
+ })  # end sapply
> ls()  # list objects
> # assign new values to objects using for loop
> for (sym_bol in load_ed) {
+   assign(sym_bol, runif(1))
+ }  # end for
> ls()  # list objects
> # save vector of objects
> save(list=load_ed, file="my_data.RData")
> # remove only loaded objects
> rm(list=load_ed)
> # remove the object "load_ed"
> rm(load_ed)
```

Saving Output of R to a File

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

Remember to call sink() to end the diversion!

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

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

The function dev.off() ends the diversion,

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

Package *googlesheets* for Interacting with *Google Sheets*

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

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

The function gs_ls() lists the files in *Google Sheets*,

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

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

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

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

You can find online a document about managing authentication tokens,

```
> # install latest version of googlesheets
> devtools::install_github("jennybc/googlesheets
> # load package googlesheets
> library(googlesheets)
> library(dplyr)
> # authenticate authorize R to view and manage
> gs_auth(new_user=TRUE)
> # list the files in Google Sheets
> googlesheets::gs_ls()
> # register a sheet
> google_sheet <- gs_title("my_data")
> # view sheet summary
> google_sheet
> # list tab names in sheet
> tab_s <- gs_ws_ls(google_sheet)
> # set curl options
> library(httr)
> httr::set_config(config(ssl_verifypeer=0L))
> # read data from sheet
> gs_read(google_sheet)
> # read data from single tab of sheet
> gs_read(google_sheet, ws=tab_s[1])
> gs_read_csv(google_sheet, ws=tab_s[1])
> # or using dplyr pipes
> google_sheet %>% gs_read(ws=tab_s[1])
> # download data from sheet into file
> gs_download(google_sheet, ws=tab_s[1],
+       to="C:/Develop/R/lecture_slides/data/goo
```

Downloading Data from *Google Sheets*

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

If you already have a *Google* account, then your personal *Google Sheets* can be found at:

https://docs.google.com/spreadsheets/

The function gs_ls() lists the files in *Google Sheets*,

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

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

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

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

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

```
> # install latest version of googlesheets
> devtools::install_github("jennybc/googlesheets
> # load package googlesheets
> library(googlesheets)
> library(dplyr)
> # authenticate authorize R to view and manage
> gs_auth(new_user=TRUE)
> # list the files in Google Sheets
> googlesheets::gs_ls()
> # register a sheet
> google_sheet <- gs_title("my_data")
> # view sheet summary
> google_sheet
> # list tab names in sheet
> tab_s <- gs_ws_ls(google_sheet)
> # set curl options
> library(httr)
> httr::set_config(config(ssl_verifypeer=0L))
> # read data from sheet
> gs_read(google_sheet)
> # read data from single tab of sheet
> gs_read(google_sheet, ws=tab_s[1])
> gs_read_csv(google_sheet, ws=tab_s[1])
> # or using dplyr pipes
> google_sheet %>% gs_read(ws=tab_s[1])
> # download data from sheet into file
> gs_download(google_sheet, ws=tab_s[1],
+       to="C:/Develop/R/lecture_slides/data/goo
```

Sourcing R Script Files in an R Session

R commands can be saved into a file, and then executed from an interactive R session using the function source(),

The function source() executes R commands contained in a file, or in a *URL*,

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

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

```
> script_dir <- "C:/Develop/R/scripts"
> # execute script file and print the commands
> source(file.path(script_dir, "script.R"),
+  echo=TRUE)
>
> ####################################
> ### script.R file contains R script to demonst
>
> # print information about this process
> print(paste0("print: This test script was run 
> cat("cat: This test script was run at:", forma
>
> # display first 6 rows of cars data frame
> head(cars)
>
> # define a function
> fun_c <- function(x) x+1
>
> # read a line from console
> readline("Press Return to continue")
>
> # plot sine function in x11 window
> x11()
> curve(expr=sin, type="l", xlim=c(-2*pi, 2*pi),
+ xlab="", ylab="", lwd=2, col="orange",
+ main="Sine function")
```

Running R Processes From the Command Window

An interactive R process can be run from the command window, by simply typing the commands R or Rterm (provided that your *PATH* variable contains the directory of the R executable file),

The command R combined with the option -e can also execute R commands supplied on the command line,

For example the command:

```
R --vanilla -e head(cars) > out.txt
```

executes a single R command, and saves the output to a file,

The option vanilla instructs R to produce minimal output,

The manual *Introduction to R* provides more information about running R processes from the command window:

https://cran.r-project.org/doc/manuals/R-intro.html#Invoking-R-from-the-command-line

```
# start an interactive R process
> R

# get help about running R process
> R --help

# execute single R command and save outp
# vanilla option to produce minimal outp
> R --vanilla -e head(cars) > out.txt
```

Executing R Scripts as Batch Processes

A *batch* process is the execution of a set of commands in a script file, without manual intervention (non-interactive mode),

There are two ways of running an R script file:

- in *interactive* mode from within an R session using the function `source()`,
- in non-interactive *batch* mode from a command window,

R *batch* processes can be executed using the commands `R`, `R CMD BATCH`, and `Rscript`,

For example the command:

```
Rscript script.R > out.txt
```

executes a *batch* process on a script file containing a plot command and `readline()` for user input, and saves the output to a file,

The command `Rscript` can also execute R commands supplied on the command line, for example:

```
Rscript -e "head(cars)" > out.txt
```

```
> # get help about running R scripts and batch p
> ?BATCH
> ?Rscript
```

```
# execute script file and save output to
# vanilla option to produce minimal outp
> cd C:/Develop/R/scripts
> R --vanilla < script.R > out.txt

# execute script file and save output to
# slave option to produce minimal output
> R CMD BATCH --slave script.R out.txt

# execute script file and save output to
> Rscript script.R > out.txt

# execute single R command from Windows
> Rscript -e "head(cars)" > out.txt

# execute several R commands and save ou
> Rscript -e "source('script.R'); fun_c(
```

Executing R Scripts Using Rscript

The function commandArgs() returns a vector of strings containing the arguments supplied to the R process when called from the command line,

The Rscript command is designed for fast execution of R scripts, and can also accept arguments to the R script supplied on the command line, for example:

```
Rscript --vanilla script_args.R  4 5 6
```

The Rscript command can also accept arguments supplied to R scripts on the command line, for example:

```
Rscript -e "2*as.numeric(commandArgs(TRUE))" 3

Rscript -e "sum(as.numeric(commandArgs(TRUE)))" 4 5 6
```

```
> ### script_args.R contains R script that accep
> # print information about this process
> cat("cat: This script was run at:", format(Sys
> # read arguments supplied on the command line
> arg_s <- commandArgs(TRUE)
> # print the arguments
> cat(paste0("arguments supplied on command line
> # return sum of arguments
> sum(as.numeric(arg_s))
```

Plotting to a File From an R Script

A *batch* R process usually fails to produce a plot, because the x11 plot window closes as soon as the R process terminates,

The function readline() doesn't work in batch mode either, because it doesn't wait for user input,

But a *batch* R process can plot to a file by diverting its graphics output to a graphics file,

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

The function dev.off() ends the diversion,

```
> ### plot_to_file.R
> ### R script to demonstrate plotting to file
>
> # redirect graphics output to png file
> plot_dir <- "C:/Develop/data"
> png(file.path(plot_dir, "r_plot.png"))
>
> # plot sine function
> curve(expr=sin, type="l", xlim=c(-2*pi, 2*pi),
+ xlab="", ylab="", lwd=2, col="orange",
+ main="Sine function")
>
> # turn png output off
> dev.off()
```

```
# execute script file and save output to
> Rscript plot_to_file.R > out.txt
```

Interactive Plots in Batch R Processes

Interactive plots don't work in batch R processes, because the attached x11 plot window closes as soon as an R process terminates,

One way to get around this is by pausing the R process using a while() loop, to wait until all the x11 plot windows are closed,

The function dev.list() returns the number and names of active graphics devices,

```
> ### plot_interactive.R
> ### R script to demonstrate interactive plotti
>
> # plot sine function in x11 window
> x11()
> curve(expr=sin, type="l", xlim=c(-2*pi, 2*pi),
+ xlab="", ylab="", lwd=2, col="orange",
+ main="Sine function")
>
> # wait until x11 window is closed
> while (!is.null(dev.list())) Sys.sleep(1)
```

```
# execute script file and save output to
> Rscript plot_interactive.R > out.txt
```

Performing Calculations in *Excel* Using R

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

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

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

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

```
> ### perform calculations in R,
> ### and export to CSV files
> setwd("C:/Develop/R/lecture_slides/data")
> # read data frame, with row names from first c
> data_read <- read.csv(file="florist.csv",
+                       row.names=1)
> # subset data frame
> data_read <-
+   data_read[data_read[, "type"]=="daisy", ]
> # write data frame to CSV file, with row names
> write.csv(data_read, file="daisies.csv")
```

Running R Code from *Excel*

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

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

```
> ### perform calculations in R,
> ### and export to CSV files
> setwd("C:/Develop/R/lecture_slides/data")
> # read data frame, with row names from first c
> data_read <- read.csv(file="florist.csv",
+                       row.names=1)
> # subset data frame
> data_read <-
+   data_read[data_read[, "type"]=="daisy", ]
> # write data frame to CSV file, with row names
> write.csv(data_read, file="daisies.csv")
```

Running R Code Using *VBA* Scripts

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

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

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

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

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

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

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

BERT Excel Add-in for Running R Code

BERT is an *Excel* add-in which allows executing R commands as *Excel* functions:

http://bert-toolkit.com/
http://bert-toolkit.com/bert-quick-start

https://github.com/sdllc/Basic-Excel-R-Toolkit/wiki
https://github.com/sdllc/Basic-Excel-R-Toolkit

BERT launches its own R process from *Excel*,

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

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

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

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

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