


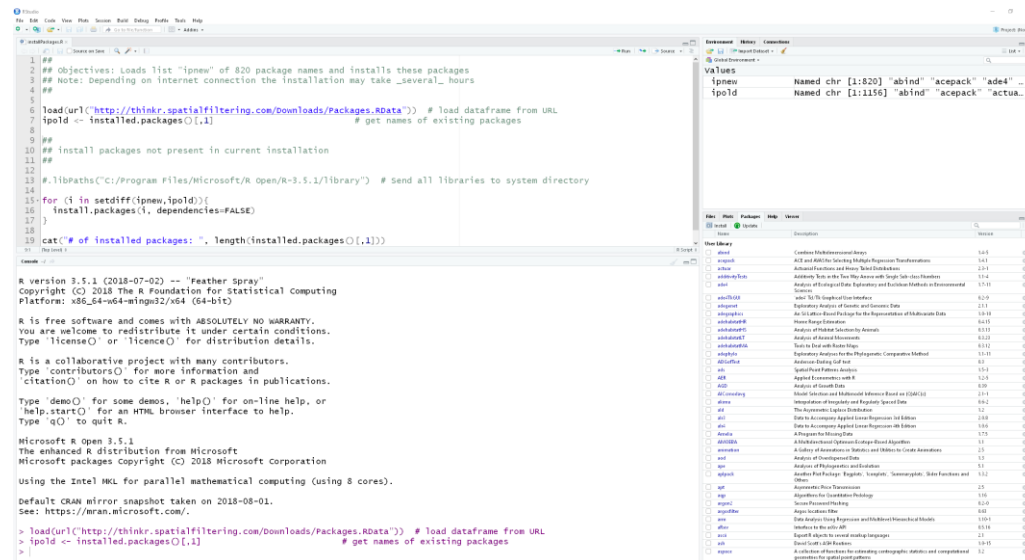





Preamble

- An excellent website outlining data operations and basic data analysis with  can be found at <http://www.statmethods.net>
- Explore Microsoft's Open R website: <http://mran.microsoft.com/>
- Explore RStudio's website: <https://www.rstudio.com/>

The RStudio Environment

- Generate a working directory, such as **Drive:\Path** on your jump drive and subsequently start .
- Discuss the  interface structure: **EDITOR – CONSOLE – ENVIRONMENT/HISTORY/CONNECTIONS – FILES/PLOTS/PACKAGES/HELP/VIEWER**




- Explore 's Tools and Help tab and 's help system.
- Concept of workspace, history file and script. The associate file extensions are:
 - ***.RData** or ***.rda**: Copy of the workspace with all its data objects and custom user functions.
 - ***.Rhistory** contains all command that have been issued during a session at the command prompt >
 - ***.R** is a file that contains scripts, which can be a set of basic  data analysis commands, individual functions, or an elaborate program
- Get and change the *working directory* where your scripts and workspace are stored and searched by default.

Enter the command `getwd()` into the editor window and select “Run” from .


Your current working directory is displayed in the **CONSOLE** below


To change the working directory using the console type `setwd("Drive:/Path")` or from the menu with **SESSION ► SET WORKING DIRECTORY ►**



Comments:


- The *string* **"Drive:/Path"** is the path to the working directory. In  strings are always enclosed by quotation marks, i.e, "...".

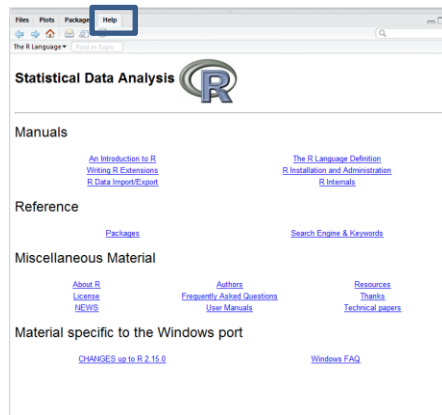
The **WINDOWS** convention is to separate sub-directories by a backward slash \.

However,  uses the forward slash / or alternatively a double backward slash \.

In  the single backslash \ is reserved to start an *escape* character, for instance, \n becomes a line break and carriage return in text output, e.g., > `cat("First line\n Second line")` is shown in the **CONSOLE** on two lines.

- The character > in the **CONSOLE** window is the command prompt which indicates that  is ready to receive new commands. It shows up when  completed executing a command or complete script.

- The **ESC** key or pressing  in the **CONSOLE** window – available while a script is executing – can terminate the ongoing execution of a script.
- The argument(s) of a function are enclosed within the parentheses. Even if no options are specified a function always ends with parentheses ().
- Receiving help at the command prompt:
 - **> help(FunctionName)** or shorter **> ?FunctionName**
This searches all *active* libraries (libraries currently linked to your session) for the help.
 - Or through the **HTML-help** menu system:

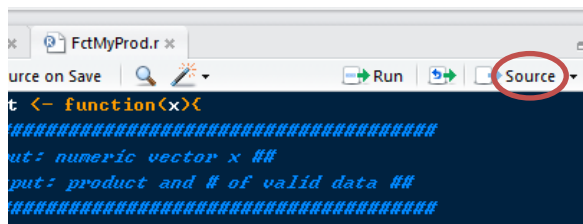



- Fuzzy help can be obtained with the double question mark **> ??PartName**. This searches all installed libraries for the help on functions, data or vignettes containing the string “PartName”.

Basic Mechanics


Interacting with the R-Console

- Within a command line, anything behind the number sign “#” is not evaluated and interpreted as a comment.
For instance, `> setwd("Drive:/Path") # Pick your working directory`
- Single commands can be run from the command prompt or with **RUN** for a highlighted line in a script.
- Collections of commands (or programs) can be stored in external ***.R** script-files (see **FILE** menu)
- To run a script (all lines in your editor) use the **SOURCE** button:






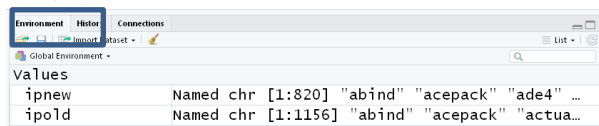
- In the **CONSOLE** window the arrow **UP** and **DOWN** keys scroll through the history of previously issued  commands.
- Previously issued commands can be edited by moving with the arrow keys the edit prompt to a particular location of your command line.
- While being in the console window, commands can be broken over several lines. Then the continuing prompt “+” will be displayed *automatically* at the beginning of a new line until the command is completed.
- To clean a cluttered **CONSOLE** window use the key combination **CTRL-L**.



Variables


- Assignment operator to a variable is the backward arrow “<-”, i.e., it requires two key-strokes,
`> my.pi <- 3.14` (compare to the  constant `pi`)

Note: The backward arrow <- is preferred over the equal sign = , which is reserved for parameter assignments in function calls.

- **Variable names:**
 - Names must begin with a letter and can consist of an alpha/numeric combination of letters including the period “.” and/or the underscore “_”
 - Note: Specific characters and keywords such “\$” “@” “&” and “%” or “+” “-” “*” “/” and “^” **cannot** be used because they have special meanings. For a full list see `> ?Reserved`
 - Warning: Variable and function names in  are *case sensitive*, e.g., `my.Var` and `my.var` are different objects (\Rightarrow this can lead to typos while writing a script, which are extremely difficult to spot)
 - Tip: Name variables properly so an external reader or you, after a few weeks have passed, can understand what you were doing
 - Use the dot to structure variable names, e.g., `sales.plano` and `sales.dallas`, or the camelback convention `salesPlano` and `salesDallas`
 - The document [GOOGLE-R-STYLE.PDF](#). suggests professional naming and typesetting conventions of your  code.
- Any *function* or *data structure* that is defined during a -session becomes an object in the environment



- These can be removed from the environment with the remove function
`> rm(My.Var)`
- To clean the everything from the environment use the command `> rm(list=ls(all=TRUE))` or the broom symbol  in the **ENVIRONMENT** menu bar.
- Warning: if you happen to name a variable or function identically to an existing  object, which already exists in the *search path* of your session, therefore, that object will be masked and is no longer directly accessible:

```
> pi                # gives the system constant 3.141593
> pi <- 2.71        # this masks the system constant pi
> base::pi          # pi still be found in its library base. Note the "::"
> rm(pi)            # removes user's pi and makes the constant pi available
```
- Some hard-wired values in  are:
 - *Logical* values **T** and **F** (alternatively **TRUE** and **FALSE** can be used)
 - An object *without content* has the value **NULL**
 - *Impossible* operations, such as `log(-1)`, lead to a *not-a-number* **NaN**.
 - *Missing* numbers have the value **NA** which stands for not available.
 - Some *predefined* numbers are infinity **Inf** and **pi** ($\pi=3.141593$)

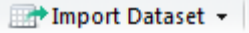
Data Representation in

Data-Sets

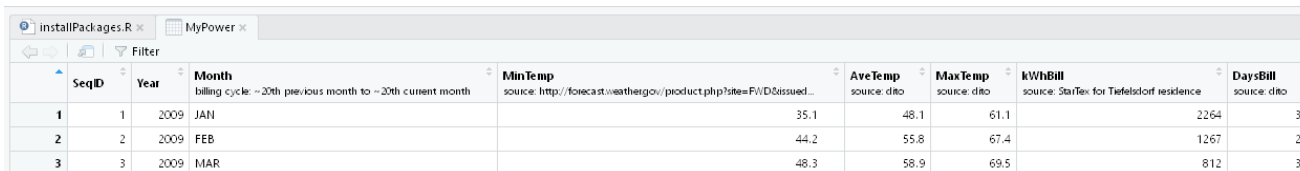
- For statistical analyses data-sets are usually arranged in rectangular data-frames and imported from external files (such as **SPSS**, **STATA**, **EXCEL** or **DBASE**) or embedded in workspaces (with the extension ***.RData**) of libraries.

For instance, to read an SPSS file use:


```
> library(foreign) # makes import functions available during a session
> setwd("E:\\Lectures2018\\WorkingWithR")
> MyPower <- read.spss("DallasTempPower.sav", to.data.frame=T)
```

- Some data file types can also be imported using **FILE ► IMPORT DATASET** or with  in the **ENVIRONMENT** window.
- Notice that the object **MyPower** is added as data-frame to your environment. Check the class of an object use


```
> class(MyPower)
[1] "data.frame"
```
- To preview the data-frame double click on it.



SeqID	Year	Month	MinTemp	AveTemp	MaxTemp	kWhBill	DaysBill
		billing cycle: ~20th previous month to ~20th current month	source: http://forecast.weather.gov/product.php?site=FW/D&issued...	source: dito	source: dito	source: StatTex for Tiefelsdorf residence	source: dito
1	2009	JAN	35.1	48.1	61.1	2264	34
2	2009	FEB	44.2	55.8	67.4	1267	29
3	2009	MAR	48.3	58.9	69.5	812	30

- Each column in a data-frame has an associated elementary data type.
- Some  libraries also include data. These can be opened by `> data("CPS1985", package="AER")`

Elementary Data Types

- Within a data-frame the variables can be of an elementary  data types:
 - logical: **FALSE** or **TRUE** (also binary **0** or **1**), e.g., `am.I.logical <- TRUE`
 - character strings: always enclosed by single or double quotation marks, e.g., `my.name <- "Michael"`.
 - factors: Internally each value is stored with a specific integer numbers. Each integer has a descriptive label assigned to it. When listed, factor values are not enclosed within quotation marks. For instance:

```
> MyPower$Month
[1] JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
[13] JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
[25] JAN FEB MAR APR MAY JUN JUL AUG
Levels: JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
```
 - numeric: either integer, double precision or complex numbers, which are not used in this course.
 - True integer values can be enforced with the added symbol “**L**”

```
> two <- 2L.
> typeof(two)
[1] "integer"
```
 - Integers, under specific circumstance, are treated as real numbers or may be *coerced* to real numbers


```
> two.sq <- two * 2.0
```

```
> typeof(two.sq)
```

```
[1] "double"
```

- Very small remainders after operations are interpreted as zeros, such as

```
> (sqrt(2))^2 - 2
```

```
[1] 4.440892e-16.
```

This discrepancy is due to *rounding errors* associated with floating point computations, i.e. taking the square root and squaring the remaining results. This problem applies to all software environments and computer chips dealing with rational numbers.

Rational numbers with an infinite number of digits can only be *digitally approximated* due to the limited bit-depth implemented in operating systems.

E.g., while the constant $\pi = 3.14 \dots$ has an infinite number of digits, only the first 16 can be numerically represented:

```
> options(digits=20)
```

```
> pi
```

```
[1] 3.1415926535897931
```

- Dates: Dates come in different formats. This course will *not* cover dates.

Basic Data Objects in :

- scalar: an individual datum. All data objects are composed of a collection of scalars.
- vector: atomic data structure that collects more than one value of identical type. Example:

```
> score <- c(23, 53, 45, 30, 53, 60)
```

 where the function `c()` concatenates – combines –

several scalars into a vector or

```
> my.cats <- c("Charlie", "Gretchen", "Austin") .
```

```
> length(score) gives the number of elements in the vector.
```

Note: if a printed vector stretches over several rows in the console the position of the first value in each row will be numbered by `[elementNumber]`

Mixture of scalars with different data types will be *coerced* into characters

```
> mixture <- c(1,2,3,T,pi,"A")
```

```
> mixture
```

```
[1] "1" "2" "3" "TRUE" "3.14159265358979" "A"
```

- matrix: two-dimensional arrangement of a set of vectors that are all of the *same data type and length*.
 - data frames: collection of vectors of the *same length* but *different data types* per column are allowed.
 - list: collection of vectors of any type and length.
- To obtain information about a data object and to look at its structure use the function `str()`:

```
> str(MyPower)
```

```
'data.frame':    32 obs. of  8 variables:
```

```
$ SeqID      : num  1 2 3 4 5 6 7 8 9 10 ...
```



```
$ Year       : num  2009 2009 2009 2009 2009 ...
```

```
$ Month      : Factor w/ 12 levels "JAN","FEB","MAR",...: 1 2 3 4 5 6 7...
```

```
$ DaysBill   : num  34 29 30 32 29 30 32 29 30 31 ...
```

```
- attr(*, "codepage")= int 1252
```

Object Oriented Philosophy

-  is an object oriented data analysis language, which is centered around functions that are applied to objects.
- All data objects have an object class assigned to them. This may be fairly rudimentary such as being numeric or highly advanced such as a spatial polygon data structure.
- All  commands are *implemented as functions* with parentheses at the end of the function name.
A function may not use [a] specific input arguments, e.g., the function `getwd()`, or [b] a selection of arguments, [c] some of them may have default values and only need to be issued when they are overwritten, and [d] some arguments are positional (without a leading keyword) or are introduced by a specific keyword.
The general syntax of a function is:

```
ResultObjectClass <- function(List of InputArgumentsClass)
```

- Depending on the class of the input arguments, an object-specific function will be used. Example:

```
> data(dataframe)           # Attach a dataframe to a session  
> summary(dataframe)        # Get summary statistics for all variables  
> regObject <- lm(...)      # run a regression model and store in regObject  
> summary(regObject)        # view key results of the regression analysis
```
- If just the object name is entered at the command prompt its *default print method* will be used. E.g.,

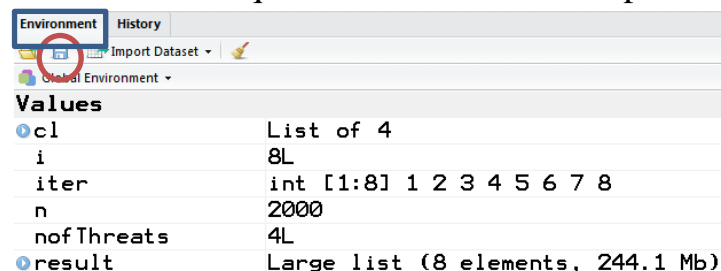
```
> regObject                 # print content of regObject on consol
```

Saving Analysis Results, Plots and Data

- Analysis Results and Plots: The easiest way of saving [a] **CONSOLE** output and [b] generated plots in the graphics window is to *copy and past* them into a graphically enhanced text editor such as **WORD**.
Important: Any text output needs to be typeset in a *fix pitch font* such as **Courier New**.

Otherwise the formatting of the output will be lost, e.g., columns of a matrix ***will not*** line up properly. Perhaps reduce the font size and line-spacing of imported output as well as switch to a landscape layout.

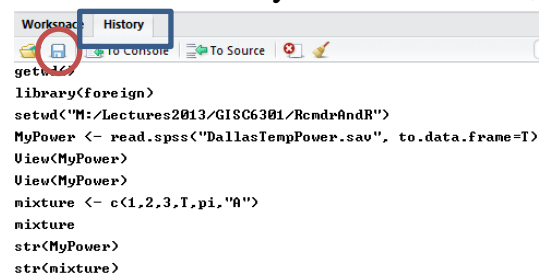
- Data: The collection of all variables, data-frames and functions, which were created during a session, can be saved for subsequent sessions in a workspace



Before saving the workspace non-desired data-objects, variables and functions should be dropped with the remove command:

```
> rm(objectName).
```

- Data-frames within the workspace can also be ***exported*** into different file formats (see the package **foreign** or connections to SQL servers).
- Command History: All commands, which were issued during session, can be saved into a history file

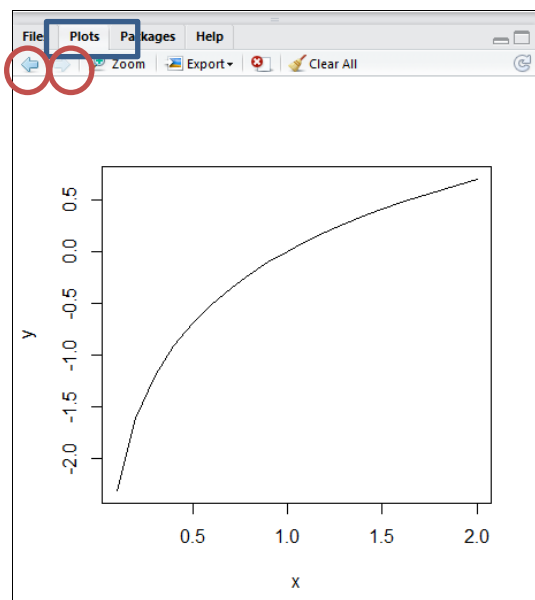


Working with some functions

- Exercise:

```
> x <- seq(0.1 ,2,by=0.1)  # sequence of numbers: 0.1,0.2,...,2
> x                        # show numbers
> y <- log(x)              # calculate the natural logarithm
> plot(x,y)               # Plot y against x
> plot(y~x)               # same plot conceiving y as a function of x
> help(plot)              # Explore options
> plot(x,y,type="l")      # Connect points by lines
```

- Select in the Plots window scroll through your list of plots with the arrow icons:



- Exercise (cont.):

```
> z <- rnorm(length(x)) # vector of standard normal random numbers
> mat <- cbind(x,y,z)    # merge vectors of same length into a matrix
> dim(mat)              # see the dimensions of the matrix
> summary(mat)          # get statistics of each matrix column
> class(mat)            # evaluate object type
[1] "matrix"
> df <- data.frame(x=x, xlog=y, rand=z) # build dataframe
> class(df)
[1] "data.frame"
```

Working with Data

Data-Frames

- Data-frames can pool several vectors of *same length* but potentially of *different data-types* together.
- Almost all statistical analysis functions are defined on data-frames.


```
> my.cats <- c("Austin","Gretchen","Charlie") # Define variables
> age.cats <- c(3,4,12)
> my.data <- data.frame(Name=my.cats, Age=age.cats) # Define data-frame
> my.data # Show data-frame
```

	Name	Age
1	Austin	3
2	Gretchen	4
3	Charlie	12

- The variables **name.cats** and **age.cats** are stored now in the data frame **my.data** under new names **Name** and **Age**
- To access individual variables in the data frame several commands can be used

```
> my.data$Name
> my.data["Name"]
> my.data[1]
> with(my.data, Name)
> my.data[, "Name"]
> my.data[, 1]
```

List Objects

- Note:  functions can only return one data object. However, different data object can be bundled into a list and returned by the function.
- List objects allow to link data objects of *different types* and *different length* together into a container:

```
> my.cats <- c("Austin", "Gretchen", "Charlie") # character vector
> age.cats <- c(2, 3, 12)                        # numeric vector
> A <- matrix(c(1, 2, 3, 4, 5, 6), 2, 3)          # 2 x 3 matrix
> my.list <- list(name = my.cats, age = age.cats, mat = A)
> my.list
$name
[1] "Austin"    "Gretchen"   "Charlie"
$age
[1]  2   3  12
```

```
$mat
```

```
      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- Individual objects of the list can be addressed either by

```
> my.list$name
```

```
[1] "Austin"    "Gretchen"    "Charlie"
```

```
> my.list[[1]]
```

```
[1] "Austin"    "Gretchen"    "Charlie"
```

- Individual elements of an object in a list can be addressed by

```
> my.list$name[1]
```

```
[1] "Austin"
```

- To delete an object in a list assign the **NULL** value:

```
> my.list$mat <- NULL
```

- To get information about an object use the `attributes` function:

```
> attributes(my.list)
```

```
$names
```

```
[1] "name" "age"
```

- Remove the objects `my.list` and the matrix `A` from the workspace

```
> rm(my.list,A)
```

Matrix Objects

- Matrices can store vectors of *same length* and *same data-type* in an rectangular arrangement

- To generate a matrix:
 - Vector of 12 elements:
`> b <- c(10,20,30,40,15,25,35,45,1,2,3,4)`
 - Place elements into 4x3 matrix:
`> mat <- matrix(b, nrow=4, ncol=3)`
- One element at location row and col `mat[row,col]`
A sequence of values `mat[1:2,]` (here the first and second row)
Exclusion of elements `mat[-1,]` (here the first row)
- One row `mat[1,]` or one column `mat[,2]`
- Also logical operations are permitted (here the first and second column):
`> select <- c(T,T,F)`
`> mat[,select]`