What is R ?

- R is a high-level computer language and environment for statistics and graphics:
  - performs a variety of simple and advanced statistical methods,
  - produces high quality graphics,
  - enables writing new functions that extends R 's uses.
- R is a free open source software maintained by several contributors. Including an "R Core Team" of several programmers who are responsible for modifying the R source code.
- The official R home page is: http://www.R-project.org
- The R system can be installed on, Windows, Mac or Linux. Visit the R website and follow the installation directions to install the base system.
- Stock and Watson textbook R companion book: https://www.econometrics-with-r.org/index.html
- Wooldridge textbook R companion book: http://www.urfie.net/index.html

QUEENS COLLEGE

What is R ?

- R is a free implementation of a dialect of the S language. A preliminary version of S was created by Bell Labs in the 1970s that was meant to be a programming language like C but for statistics.
- John Chambers was one of the primary inventors for the language, and he won the Association for Computing Machinery Award in 1999.
- S is a high-level programming language, with similarities to Python. By the 1980s, the popularity of S was growing outside of Bell Labs. This led to Statistical Sciences Inc. creating the S-Plus software package based on S in 1988 and then selling it to users. In 1993, Statistical Sciences Inc. merges with a company named Mathsoft and buys the exclusive license to market S. This prevents anyone from obtaining a free version from Bell Labs. After further company merges, S-Plus now resides with Tibco Software Inc.
- R was initially written by Ross Ihaka and Robert Gentleman at the Department of Statistics of the University of Auckland in Auckland, New Zealand (hence the name).

QUEENS COLLEGE

What is R ?

- Version 1.0.0 of R was released on February 29, 2000. Since this time, the use of R has skyrocketed.
- In addition to the base system there are user contributed add-on packages, i.e., collection of functions, examples and documentation that usually focus on a specific task.
- The base system contains some packages. To install an additional package, say xtable, be connected to the Internet and type

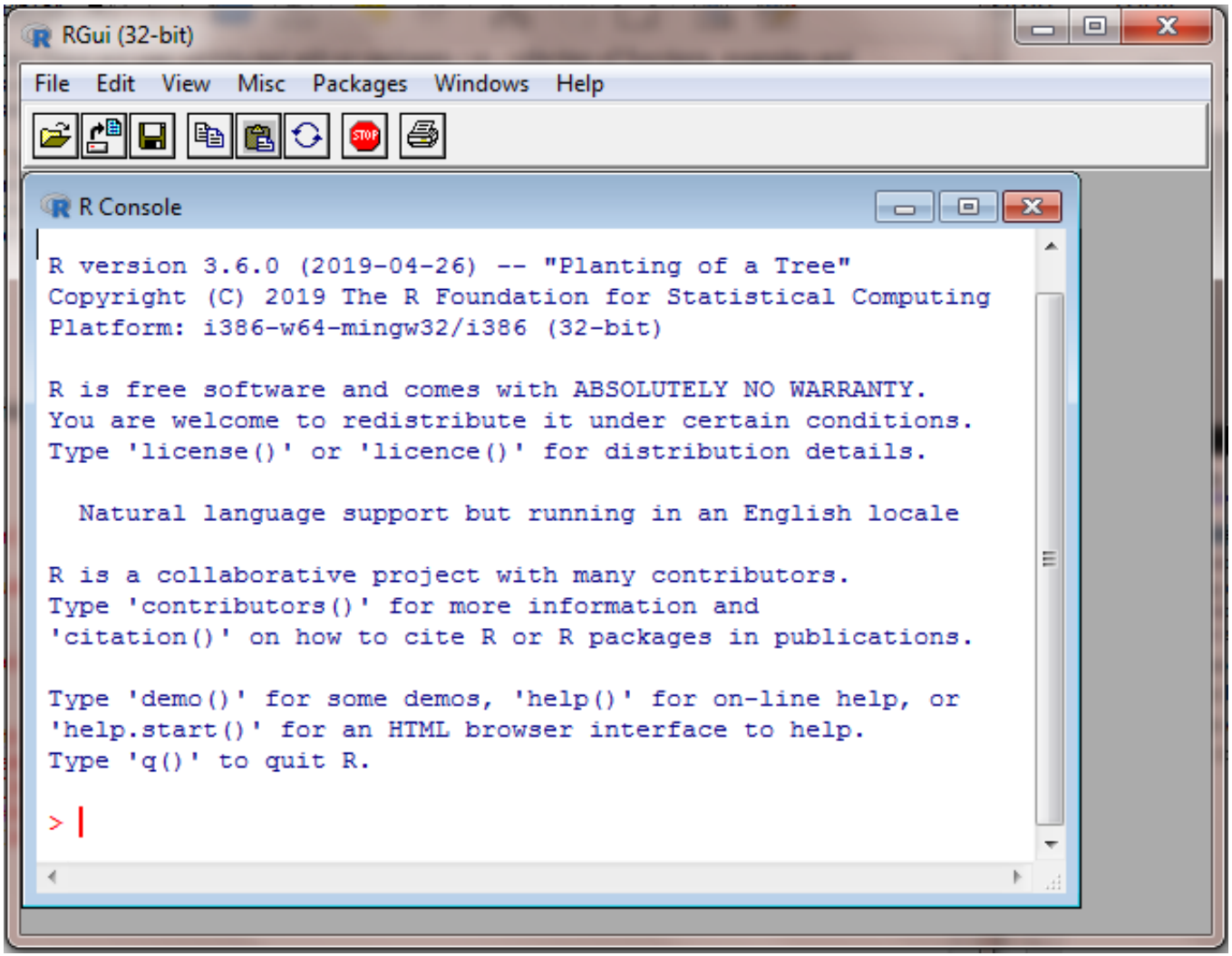
  ```
  install.packages("xtable")
  ```

  You will be asked to select the mirror site nearest to you, after that everything is automatic.
- In order to use the contents of the package we need to load it,

  `library(xtable)` or `require(xtable)`

R Console

- The R Console is where computations are performed.



R as a calculator

- You can use R as a calculator. Enter a math expression and the result is outputted to the console.

Binary Operations	+	-	*	/	^	%%
Math Functions	abs	sqrt	log	exp	log10	factorial
Trig Funtions	sin	cos	tan	asin	acos	atan
Rounding	round	ceiling	floor	trunc	signif	zapsmall
Math Quantities	Inf	-Inf	NaN	pi	exp(1)	1i

```
> 5+5*(1+0.4)
[1] 12
> pnorm(1.96)
[1] 0.9750021
> (2-3)/6
[1] -0.1666667
> 2^2
[1] 4
> sin(pi/2)
[1] 1
> log(4)
[1] 1.386294
```

R : Objects and Functions

- What we normally do is create objects and then perform functions on those objects (functions are also considered objects).
- Assign an object a name `"x"` using either

  `x <-` *object* (*object* `->x`)
  `x =` *object*

- Call a function by

  function name(list of arguments separated by commas)

  - Each function has a set of formal arguments some with default values; these can be found in the function's documentation.
  - To specify a particular argument use the argument's name.

- R is `CASE SENSITIVE`.

R : Objects and Functions

- Suppose I want to find the mean of a set of numbers. I first assign the vector of numbers to `x` and then call the function `mean()`.

```
> x=c(0,5,7,9,1,2,8)
> x
[1] 0 5 7 9 1 2 8
> mean(x)
[1] 4.571429
> ls()
[1] "x"
> objects()
[1] "x"
```

- Suppose I want to sort a vector `y` so that the numbers are descending.

```
> y=c(4,2,0,9,5,3,10)
> y
[1]  4  2  0  9  5  3 10
> sort(y)
[1]  0  2  3  4  5  9 10
> sort(y,decreasing=TRUE)
[1] 10  9  5  4  3  2  0
```

R : Objects and Functions

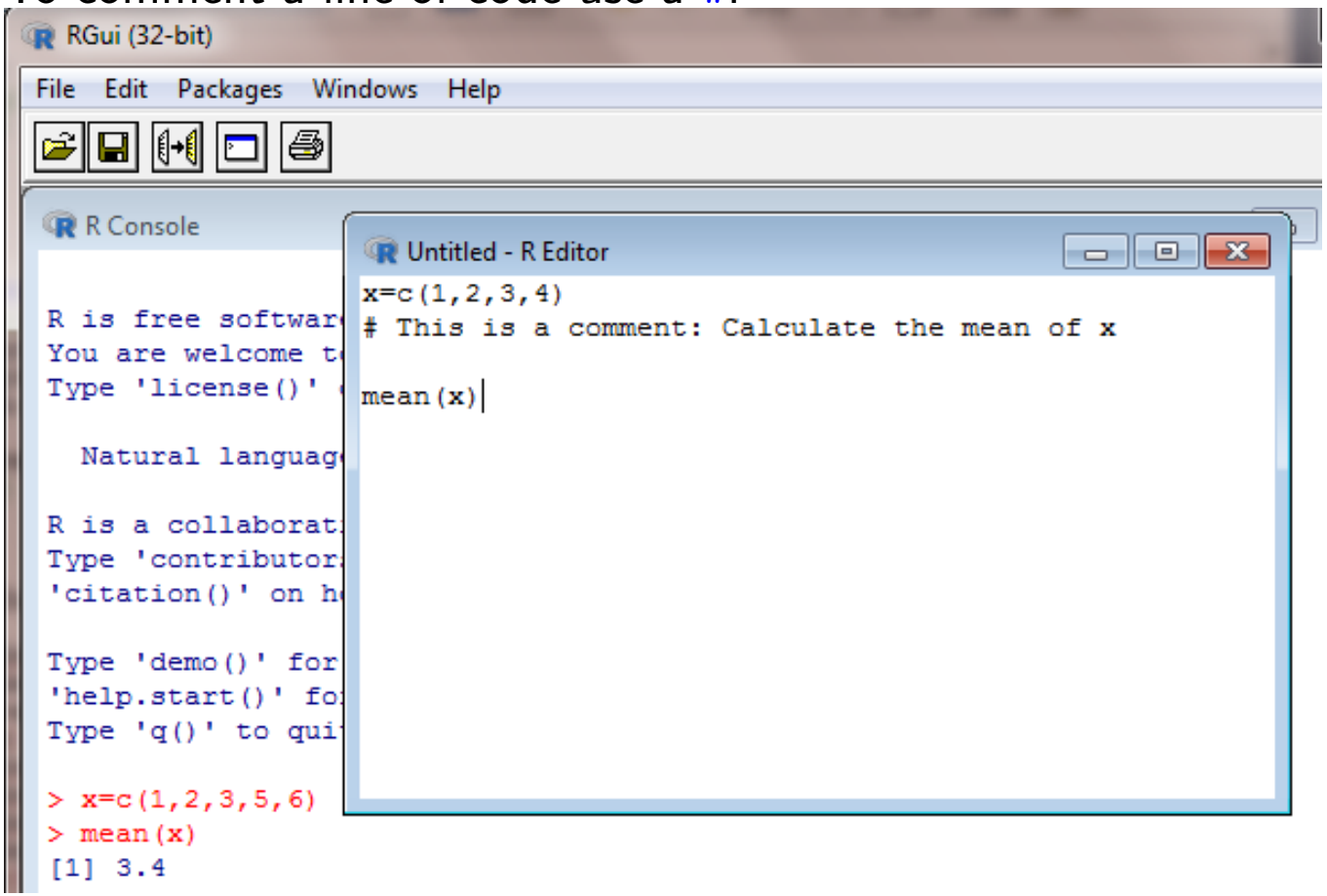
- A function `mysd()` can be written to find the standard deviation of a given vector of numbers.

```
# Declaring a function
mysd<-function(numbers) {
  sqrt(var(numbers))
}

> x<-c(1,2,3,4,5)
# Call the function "mysd()"
> mysd(x)
[1] 1.581139
```

R : Script Editor

- The script editor is used for writing programs in R .
- Starting with R 2.0, a VERY limited program editor was incorporated into R.
  To open script editor select `File > New script` to create a new program.
- To comment a line of code use a `#`.



R : Script Editor

- Instead of this editor, we will use RStudio. This software is available for free, and it runs on Linux, Mac, and Windows operating systems.
- This software is actually more than an editor because it integrates a program editor with the R Console window, graphics window, R-help window, and other items within one overall window environment. Thus, RStudio is an integrated development environment (IDE) for constructing and running R programs.
- RStudio can be downloaded at `http://www.rstudio.com`.

Documentation

- To look at the documentation for a specific function from a loaded package,
  ?function name
  `help(`*function name*`)`
- For a complete list of functions, use library(help = package name).
- To run the example included with the documentation, `example(`*function name*`)`.
- To get an overview of an installed package and its contents, help(package=package name).
- Some packages include a vignette, a file with additional documentation and examples. To get a list of all vignettes from all installed packages, type vignette(). To see a particular vignette, vignette("topic").

```
?mean;     ?pnorm;     help("mean");     help("pnorm");
example("mean");          library(help="stats");
vignette();    vignette("ggplot2-specs")
```

R : Working Directory & Workspace

- To set the working directory click `Session > Set Working Directory > Choose Directory` ... or type `setwd(`*file path*`)`.

```
> setwd("C:/Users/????/Desktop/R_intro/R_intro")
> getwd()
[1] "C:/Users/????/Desktop/R_intro/R_intro"
```

- The workspace is where all the objects you create during a session are located.
- When you close R you will be prompted to "`Save workspace image?`" If you say yes, the next time you open R from the same working directory the workspace will be restored.
- That is, all of the objects you created during your last session will still be available.

R : Managing the Workspace

- To list what variables you have created in the current session, `ls()`
- To see what libraries and dataframes are loaded, `search()`
- To see what libraries have been installed, `library()`
- To remove an object, `rm(`*object names*`)`
- To remove all objects, `rm(list = ls())`

```
ls() #list of objects created in the current session
search() #loaded libraries and dataframes
library() #installed libraries

xdata=c(2,30,1,2)
rm(xdata)

rm(list=ls()) # remove all objects from the current session
```

Vectors

- Data structures
  - Vectors
  - Matrices
  - Arrays
  - Lists
  - Dataframes
- Data types
  - Numeric
  - Logical
  - Character
  - Factor
  - Dates
  - Missing data

Vectors

- A vector is an ordered collection of objects of the same type.
- The function `c(...)` concatenates its arguments to form a vector.
- To create a patterned vector
  - `:` Sequence of integers
  - `seq()` General sequence
  - `rep()` Vector of replicated elements

```
v1 <- c(2.5, 4, 7.3, 0.1)
v1
[1] 2.5 4.0 7.3 0.1

v2 <- c("A", "B", "C", "D")
v2
[1] "A" "B" "C" "D"

v3 <- -3:3
v3
[1] -3 -2 -1 0 1 2 3
```

```
seq(0,2,by=0.5)
[1] 0.0 0.5 1.0 1.5 2.0

seq(0,2,len=6)
[1] 0.0 0.4 0.8 1.2 1.6 2.0

rep(1:5,each=2)
[1] 1 1 2 2 3 3 4 4 5 5

rep(1:5,times=2)
[1] 1 2 3 4 5 1 2 3 4 5
```

Reference Elements of a Vector

- Use [ ] with a vector/scalar of positions to reference elements of a vector.
- Include a minus sign before the vector/scalar to remove elements.

```
> x <- c(4, 7, 2, 10, 1, 0)
> x[4] # return the fourth element
[1] 10
> x[1:3] # return the first three elements
[1] 4 7 2
> x[c(2,5,6)] # return elements with index c(2,5,6)
[1] 7 1 0
> x[-3] # remove the third element
[1]  4  7 10  1  0
> x[-c(4,5)] # remove the the fourth and fifth element
[1] 4 7 2 0
> x[x>4] # return elements bigger than 4
[1]  7 10
> x[3] <- 99 # change the third element to 99
> x
[1]  4  7 99 10  1  0
```

which() and match()

- Additional functions that return the indices of a vector:
  - `which()` Indices of a logical vector the condition is `TRUE`
  - `which.max()` Location of the (first) maximum element of a numeric vector
  - `which.min()` Location of the (first) minimum element of a numeric vector
  - `match()` First position of an element in a vector

```
x <- c(4, 7, 2, 10, 1, 0)
x>=4
[1] TRUE TRUE FALSE TRUE FALSE FALSE
which(x>=4)
[1] 1 2 4
which.max(x)
[1] 4
x[which.max(x)]
[1] 10
max(x)
[1] 10
```

```
y <- rep(1:5, times=5:1)
y
[1] 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
match(1:5, y)
[1] 1 6 10 13 15
match(unique(y), y)
[1] 1 6 10 13 15
```

Useful Vector Functions

`sum(x)`	`prod(x)`	Sum/product of the elements of `x`
`cumsum(x)`	`cumprod(x)`	Cumulative sum/product of the elements of `x`
`min(x)`	`max(x)`	Minimum/maximum element of `x`
`mean(x)`	`median(x)`	Mean/median of `x`
`var(x)`	`sd(x)`	Variance/standard deviation of `x`
`cov(x,y)`	`cor(x,y)`	Covariance/correlation of `x` and `y`
`range(x)`		Range of `x`
`quantile(x)`		Quantiles of `x` for the given probabilities
`fivenum(x)`		Five number summary of `x`
`length(x)`		Number of elements in `x`
`unique(x)`		Unique elements of `x`
`rev(x)`		Reverse the elements of `x`
`sort(x)`		Sort the elements of `x`
`which(x)`		Indices of `TRUE`s in a logical vector
`which.max(x)`	`which.min(x)`	Index of the max/min element of `x`
`match(x)`		First position of an element in a vector
`union(x,y)`		Union of `x` and `y`
`intersect(x,y)`		Intersection of `x` and `y`
`setdiff(x,y)`		Elements of `x` that are not in `y`
`setequal(x,y)`		Do `x` and `y` contain the same elements?

Matrices

- To create a matrix

```
matrix(data=NA, nrow=1, ncol=1, byrow = FALSE, dimnames = NULL)
```

`data`	a vector that gives data to fill the matrix; if data does not have enough elements to fill the matrix, then the elements are recycled
`nrow`	desired number of rows
`ncol`	desired number of columns
`byrow`	if `FALSE` (default) matrix is filled by columns, otherwise by rows
`dimnames`	(optional) list of length 2 giving the row and column names respectively, list names will be used as names for the dimensions

```
x=matrix(c(5,0,6,1,3,5,9,5,7,1,5,3), nrow=3, ncol=4, byrow=TRUE,
              dimnames=list(rows=c("r.1", "r.2", "r.3"),
                            cols=c("c.1", "c.2", "c.3", "c.4")))
```

```
 x
     cols
rows  c.1 c.2 c.3 c.4
  r.1   5   0   6   1
  r.2   3   5   9   5
  r.3   7   1   5   3
```

Reference Elements of a Matrix

- Reference matrix elements using the [ ] just like with vectors, but now with 2-dimensions

```
x=matrix(c(5,0,6,1,3,5,9,5,7,1,5,3), nrow=3, ncol=4, byrow=TRUE)
x
     [,1] [,2] [,3] [,4]
[1,]    5    0    6    1
[2,]    3    5    9    5
[3,]    7    1    5    3

x[2,3] # Row 2, Column 3
[1] 9
x[1,] # Row 1
[1] 5 0 6 1
x[,2] # Column 2
[1] 0 5 1
x[c(1,3),] # Rows 1 and 3, all Columns
     [,1] [,2] [,3] [,4]
[1,]    5    0    6    1
[2,]    7    1    5    3
```

Matrix operations

- When matrices are used in math expressions the operations are performed element by element.
- For matrix multiplication use the %*% operator
- Using %*% on two vectors will return the inner product as a matrix and not a scalar. Use `c()` or `as.vector()` to convert to a scalar.

```
A<-matrix(1:4, nrow=2)
B<-matrix(1, nrow=2, ncol=2)
A*B
A%*%B
     [,1] [,2]
[1,]    4    4
[2,]    6    6
y <- 1:3
y%*%y # returns dot product in matrix form
     [,1]
[1,]   14
A/(y%*%y)
Error in A/(y %*% y) : non-conformable arrays
A/c(y%*%y)
             [,1]      [,2]
[1,] 0.07142857 0.2142857
[2,] 0.14285714 0.2857143
```

Useful Matrix Functions

`t(A)`	Transpose of `A`
`det(A)`	Determinant of `A`
`solve(A,b)`	Solves the equation `Ax=b` for `x`
`solve(A)`	Matrix inverse of `A`
`MASS::ginv(A)`	Generalized inverse of `A` (MASS Package)
`eigen(A)`	Eigenvalues and eigenvectors of `A`
`chol(A)`	Cholesky factorization of `A`
`diag(n)`	Creates `n` $\times$ `n` identity matrix
`diag(A)`	Returns the diagonal elements of a matrix `A`
`diag(x)`	Create a diagonal matrix from a vector `x`
`lower.tri(A), upper.tri(A)`	Matrix of logicals indicating lower/upper triangular matrix
`apply`	Apply a function to the margins of a matrix
`rbind(...)`	Combines arguments by rows
`cbind(...)`	Combines arguments by columns
`dim(A)`	Dimensions of `A`
`nrow(A), ncol(A)`	Number of rows/columns of `A`
`colnames(A), rownames(A)`	Get or set the column/row names of `A`
`dimnames(A)`	Get or set the dimension names of `A`

Arrays

- An array is a multi-dimensional generalization of a vector
- To create an array

```
array(data = NA, dim = length(data), dimnames = NULL)
```

`data`	A vector that gives data to fill the array; if data does not have enough elements to fill the array, then the elements are recycled
`dim`	Dimension of the array, a vector of length one or more giving the maximum indices in each dimension
`dimnames`	Name of the dimensions, list with one component for each dimension, either `NULL` or a character vector of the length given by dim for that dimension. The list can be named, and the list names will be used as names for the dimensions.

- Values are entered by columns
- Like with vectors and matrices, when arrays are used in math expressions the operations are performed element by element.
- Also like vectors and matrices, the elements of an array must all be of the same type (numeric, character, logical, etc.)

Arrays

- Sample $2 \times 3 \times 2$ array

```
w <- array(1:12, dim=c(2,3,2)
        dimnames=list(c("A","B"), c("X","Y","Z"), c("N","M")))
w
, , N

  X Y Z
A 1 3 5
B 2 4 6

, , M

  X  Y  Z
A 7  9 11
B 8 10 12
```

- You can think of `"N","M"` as page numbers. Elements of an array can be referenced using `[]`.

Arrays

```
w[2,3,1] # Row 2, Column 3, Matrix 1
[1] 6

w[,"Y",] # Column named "Y"
  N  M
A 3  9
B 4 10

w[1,,] # Row 1
  N  M
X 1  7
Y 3  9
Z 5 11

w[1:2,,"M"] # Rows 1 and 2, Matrix "M"
  X  Y  Z
A 7  9 11
B 8 10 12
```

Lists

- A list is a general form of a vector, where the elements don't need to be of the same type or dimension.
- The function `list(...)` creates a list of the arguments.
- Arguments have the form *name*=*value*. Arguments can be specifed with and without names.

```
x <- list(num=c(1,2,3), "Nick", identity=diag(2))
x
$num
[1] 1 2 3

[[2]]
[1] "Nick"

$identity
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

- Elements of a list can be referenced using `[]` as well as `[[]]` or `$`.

Lists

```
y=list(v1=c(2,3,5),"Econometrics",A=matrix(1:4, nrow=2))

y[[2]] # Second element of y
[1] "Econometrics"

y[["v1"]] # Element named "v1"
[1] 2 3 5

y$A # Element named "A"
     [,1] [,2]
[1,]    1    3
[2,]    2    4

y[[3]][1,] # First row of the third element
[1] 1 3

y[1:2] # Create a sub-list of the first two elements
$v1
[1] 2 3 5

[[2]]
[1] "Econometrics"
```

Logical

- Logical values are represented by the reserved words `TRUE` and `FALSE` in all caps or simply `T` and `F`.

`!x`	NOT x
`x & y`	x and y elementwise, returns a vector
`x && y`	x and y returns a sigle value
`x | y`	x OR y elementwise, returns a vector
`x || y`	x OR y returns a sigle value
`xor(x,y)`	Exclusive OR of x and y, elementwise
`x %in% y`	x IN y
`x < y`	$x < y$
`x > y`	$x > y$
`x <= y`	$x \leq y$
`x >= y`	$x \geq y$
`x == y`	$x = y$
`x != y`	$x \neq y$
`isTRUE(x)`	TRUE if x is TRUE
`all(...)`	TRUE if all arguments are TRUE
`any(...)`	TRUE if at least one argument is TRUE
`identical(x,y)`	Safe and reliable way to test two objects for being EXACTLY e
`all.equal(x,y)`	Test if two objects are NEARLY equal

Logical

```
x <- 1:10
(x%%2==0) | (x>5) # What elements of x are even or greater than 5
[1] FALSE TRUE FALSE TRUE FALSE TRUE TRUE TRUE TRUE TRUE

y <- 5:15 # What elements of x are in y
x %in% y
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE

x[x %in% y]
[1] 5 6 7 8 9 10

any(x>5) # Are any elements of x greater then 5?
[1] TRUE

all(x>5) # Are all elements of x greater then 5?
[1] FALSE
```

Isn't that equal?

- In general, logical operators may not produce a single value and may return an `NA` if an element is `NA` or `NaN`.
- If you must get a single `TRUE` or `FALSE`, such as with if expressions, you should NOT use `==` or `!=`. Unless you are absolutely sure that nothing unusual can happen, you should use the `identical()` function instead.
- `identical()` only returns a single logical value, `TRUE` or `FALSE`, never `NA`.

```
name <- "Nick"
if(name=="Nick") TRUE else FALSE
[1] TRUE
```

- What if name is never set to "Nick"?

```
name <- NA
if(name=="Nick") TRUE else FALSE
Error in if (name=="Nick") TRUE else FALSE :
missing value where TRUE/FALSE needed

if(identical(name, "Nick")) TRUE else FALSE
[1] FALSE
```

Isn't that equal?

- With `all.equal()` objects are treated as equal if the only difference is probably the result of inexact foating-point calculations. Returns `TRUE` if the mean relative difference is less then the specifed tolerance.
- `all.equal()` either returns `TRUE` or a character string that describes the difference. Therefore, do not use `all.equal()` directly in if expressions, instead use with `isTRUE()` or `identical()`.

```
(x <- sqrt(2))
[1] 1.414214
x^2
[1] 2
x^2==2
[1] FALSE
all.equal(x^2,2)
[1] TRUE
all.equal(x^2,1)
[1] "Mean relative difference:  0.5"
isTRUE(all.equal(x^2,1))
[1] FALSE
```

Character

- Character strings are denoted by quotation marks, single ' ' or double " "

`cat()`	Concatenate objects and print to console(\n new line)
`paste()`	Concatenate objects and return a string
`print()`	Print an object
`substr()`	Extract or replace substrings in a character vector
`strtrim()`	Trim character vectors to specified display widths
`strsplit`	Split elements of a character vector according to a substring
`grep()`	Search for matches to a pattern within a character vector
`grepl()`	Like grep(), but returns a logical vector
`agrep()`	Similar to grep(), but searches for approximate matches
`regexpr()`	Similar to grep(), but returns the position of the first instance of a pattern within a string
`gsub()`	Replace all occurrences of a pattern with a character vector
`sub()`	Like gsub(), but only replaces the first occurrence
`tolower(), toupper()`	Convert to all lower/upper case
`noquote()`	Print a character vector without quotations
`nchar()`	Number of characters
`letters, LETTERS`	Built-in vector of lower and upper case letters

Character

```
animals=c("bird", "horse", "fish")
home=c("tree", "barn", "lake")

length(animals) # Number of strings
nchar(animals) # Number of characters in each string

cat("Animals:", animals) # Need \n to move cursor to a newline
cat(animals, home, "\n") # Joins one vector after the other

paste(animals, collapse=" ")#Create one long string of animals
a.h=paste(animals,home,sep=".")#Pairwise joining of animals and home

# Split strings at ".", fixed=TRUE since "." is used
# for pattern matching
unlist(strsplit(a.h, ".", fixed=TRUE))
substr(animals, 2, 4) # Get characters 2-4 of each animal
strtrim(animals, 3) # Print the first three characters
toupper(animals) # Print animals in all upper case
```

Factor

- A factor is a categorical variable with a defined number of ordered or unordered levels.

```
x=factor(rep(1:3, 4),labels=c("low","med","high"),ordered=TRUE)
x
[1] low  med  high low  med  high low  med  high low  med  high
Levels: low < med < high
```

`levels(x)`	Retrieve or set the levels of x
`nlevels(x)`	Return the number of levels of x
`relevel(x, ref)`	Levels of x are reordered so that the level specified by `ref` is first
`reorder()`	Reorders levels based on the values of a second variable
`gl()`	Generate factors by specifying the pattern of their levels
`cut(x, breaks)`	Divides the range of x into intervals (factors) determined by breaks

```
> nlevels(x)
[1] 3
> levels(x)
[1] "low"  "med"  "high"
> is.factor(x)
[1] TRUE
> is.ordered(x)
[1] TRUE
```

Statistical Distributions

- R has several built-in statistical distributions and for each distribution four functions are available:
  - `r` Random number generator
  - `d` Density function
  - `p` Cumulative distribution function
  - `q` Quantile function
- Each letter can be added as a prefix to any of the R distribution names.

R	Distribution	R	Distribution
`beta`	Beta	`logis`	Logistic
`binom`	Binomial	`nbinom`	Negative binomial
`cauchy`	Cauchy	`norm`	Normal
`exp`	Exponential	`pois`	Poisson
`chisq`	Chi-square	`signrank`	Wilcoxon signed rank statistic
`f`	Fisher's F	`t`	Student's t
`gamma`	Gamma	`unif`	Uniform
`geom`	Geometric	`weibull`	Weibull
`hyper`	Hypergeometric	`wilcox`	Wilcoxon rank sum
`lnorm`	Lognormal		

Statistical distributions

```
> dnorm(1.96, mean=0, sd=1) # Density
[1] 0.05844094
> pnorm(1.96, mean=0, sd=1) # Distribution (lower tail)
[1] 0.9750021
# Distribution(upper tail)
> pnorm(1.96,mean=0,sd=1,lower.tail=FALSE)
[1] 0.0249979
> qnorm(0.975, mean=0, sd=1) # Quantile
[1] 1.959964
> rnorm(5, mean=0, sd=1) # Five random numbers from N(0,1)
[1] -0.1921395  1.0880367  1.2215088  1.2261773 -0.2187957

> set.seed(17632)
> runif(5)
[1] 0.03288684 0.88861821 0.21466744 0.32907126 0.78222193
> set.seed(17632)
> runif(5)
[1] 0.03288684 0.88861821 0.21466744 0.32907126 0.78222193
```

Control Structures

- See `?Control` for the R documentation on control structures.
  - `if()...else` Determine which set of expressions to run depending on whether or not a condition is `TRUE`
  - `ifelse()` Do something to each element in a data structure depending on whether or not a condition is `TRUE` for that particular element
  - `switch()` Evaluate different expressions depending on a given value
  - `for()` Loop for a fixed number of iterations
  - `while()` Loop until a condition is `FALSE`
  - `repeat` Repeat until iterations are halted with a call to break
  - `break` Break out of a loop
  - `next` Stop processing the current iteration and advance the looping index

if()...else

- The condition needs to evaluate to a single logical value.
- Brackets `{}` are not necessary if you only have one expression and/or the `if()...else` statement are on one line.
- To avoid a syntax error, you should NOT have a newline between the closing bracket of the `if` statement and the `else` statement.

`if`(*condition*) *expression if TRUE*

```
if(condition) {
  expression if TRUE
}
```

`if`(*condition*) *expression if TRUE* `else` *expression if FALSE*

```
if(condition) {
  expression if TRUE
}else{
  expression if FALSE
}
```

Example `if()...else`

- Calculate the median of a random sample $X_1, \ldots, X_n$

$$\text{median}(X) = \begin{cases} \frac{1}{2}X_{(\frac{n}{2})} + \frac{1}{2}X_{(1+\frac{n}{2})} & \text{if } n \text{ is even} \\ X_{(\frac{n+1}{2})} & \text{if } n \text{ is odd} \end{cases}$$

  where $X_{(1)}, \ldots, X_{(n)}$ denote order statistics.

```
x <- c(5,4,2,8,9,10)

n <- length(x)
sort.x <- sort(x)

if(n%%2==0){
  median <- (sort.x[n/2]+sort.x[1+n/2])/2
} else {
  median <- sort.x[(n+1)/2]
}

median
```

Example: `ifelse()`

- `ifelse()` returns a value with the same structure as test which is filled with elements selected from either yes or no depending on whether the element of test is `TRUE` or `FALSE`.

```
(x <- seq(0,2,len=6))
[1] 0.0 0.4 0.8 1.2 1.6 2.0

ifelse(x <= 1, "small", "big")
[1] "small" "small" "small" "big" "big" "big"

(y <- matrix(1:8, nrow=2))
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

ifelse(y>3 & y<7, 1, 0)
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    1    1    0
```

Example: `switch()`

- The `switch()` function evaluates and returns different expressions depending on the value of `expr`: `switch(expr, ...)`.
- `expr` needs to be a single character string or a number, not a vector.
- `...` is a comma separated list of name=expression or `number=expression`. If there is no `expression`, then the next non-missing `expression` is used.

```
central <- function(y, measure) {
  switch(measure,
         Mean = ,
         mean = mean(y),
         median = median(y),
         geometric = prod(y)^(1/length(y)),
         "Invalid Measure")
}
```

```
> y=runif(100)
> central(y, "mean")
[1] 0.4844221
> central(y, "Mean")
[1] 0.4844221
> central(y, "Median")
[1] "Invalid Measure"
> central(y, "geometric")
[1] 0.3490125
```

Example: `for` loops

```
for (var in seq) {
expressions
}
```

- `var` is the name of the loop variable that changes with each iteration.
- With each iteration `var` takes on the next value in `seq`. At the end of the loop `var` will equal the last value of `seq`. The number of iterations equals the length of `seq`.
- Simulating AR(2): $y_t = \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + u_t$, where $\alpha_1 = 0.9$, $\alpha_2 = -0.2$ and $u_t \sim N(0, 1)$.

```
# Simulating AR(2) model
n=1000
u=rnorm(n)
y=rep(0,n)
for(i in 3:1000){
  y[i]=0.9*y[i-1]-0.2*y[i-2]+u[i]
}
y=y[501:1000] #take the last 500 observation
plot(y,type="l")
```

Example: `while` loops

```
while (cond) {
expressions
}
```

- cond is a single logical value that is not `NA`.

```
# Calculate 10! using a while loop
i <- 10
f <- 1
while(i>1) {
  f <- i*f
  i <- i-1
  cat(i, f, "\n")
}
```

```
> f
[1] 3628800
> factorial(10)
[1] 3628800
```

Example: `repeat` loops

```
repeat {
expressions
if(cond) break
}
```

- The `repeat` loop does not contain a limit. Therefore it is necessary to include an `if` statement with the `break` command to make sure you do not have an infinite loop.

```
# Calculate 10! using a repeat loop
i <- 10
f <- 1
repeat{
  f <- i*f
  i <- i-1
  cat(i, f, "\n")
  if(i<1){
    break
  }
}
```

```
> f
[1] 3628800
> factorial(10)
[1] 3628800
```

Vectorizing computations

- Functions that perform whole object computations or vectorizing computations can improve efficiency.

`lapply(), sapply()`	Apply a function to each element of a vector or list
`mapply()`	Multivariate version of `sapply()`, apply a function to the corresponding elements from multiple vectors
`Vectorize()`	Return a new function that acts as if `mapply()` was called

- These approaches may not improve efficiency because they do not reduce the number of function calls.

sapply()

```
poisson <- function(x, lambda){
        exp(-lambda)*lambda^x/factorial(x)
}
poisson(1:5, lambda=1)
# Suppose lambda=1,2,3,3,4,5. We can use a loop
result=matrix(nrow=5, ncol=5,
dimnames = list(rows=c("x=1","x=2","x=3","x=4","x=5"),
cols=c("lam=1","lam=2","lam=3","lam=4","lam=5")))
for(lambda in seq(1:5)){
  result[,lambda]=poisson(1:5,lambda)
}
result
     cols
rows         lam=1      lam=2     lam=3      lam=4      lam=5
  x=1 0.367879441 0.27067057 0.1493612 0.07326256 0.03368973
  x=2 0.183939721 0.27067057 0.2240418 0.14652511 0.08422434
  x=3 0.061313240 0.18044704 0.2240418 0.19536681 0.14037390
  x=4 0.015328310 0.09022352 0.1680314 0.19536681 0.17546737
  x=5 0.003065662 0.03608941 0.1008188 0.15629345 0.17546737

# Alternatively, we can use sapply()
sapply(c(lam1=1,lam2=2,lam3=3,lam4=4,lam5=5), poisson, x=1:5)
```

mapply()

```
# Evaluating gamma density
gamma.density <- function(x, alpha, beta) {
  x^(alpha-1)*exp(-x/beta)/gamma(alpha)/beta^alpha
}

# Evaluate the gamma density at 1:5 for
# alpha=beta=1, alpha=beta=2, alpha=beta=3
mapply(gamma.density, alpha=1:3, beta=1:3, MoreArgs=list(x=1:5))

# Evaluate the gamma density at 1:5 for
# all combinations of alpha and beta
parm <- expand.grid(alpha=1:3, beta=1:3)
out <- mapply(gamma.density, parm[,1], parm[,2],
        MoreArgs=list(x=1:5))
colnames(out)=paste("alpha",parm[,1],"beta",parm[,2],sep=".")
out
```

apply()

```
> # apply()
> A <- matrix(sample(1:100, 12), nrow=3, ncol=4)

> # Calculate the mean of each column
> apply(A, 2, FUN=mean)
[1] 43.66667 58.00000 52.33333 52.33333

> # Simply functions can be defined within the apply() command
> # Calculate variance of each column
> apply(A, 2, FUN=function(x) sum((x-mean(x))^2)/(length(x)-1))
[1] 2012.3333 1764.0000  192.3333  569.3333
```

Functions

- To declare a function:

```
function name <- function(argument list) {
body
}
```

- The argument list is a comma-separated list of formal argument names: (i) `name`, (ii) `name=default value`, (iii) `...`.
- The `...` is a list of the remaining arguments that do not match any of the formal arguments.
- Generally, the `body` is a group of R expressions contained in curly brackets {}. If the `body` is only one expression the curly brackets are optional.

```
myfunction<-function(numbers) {
  x1=sd(numbers) # calculate sd
  x2=mean(numbers) # calculate mean
  x3=median(numbers) # calculate mean
  out=list(mean=x2,sd=x1,median=x3)
  return(out)
}
x=c(3,4,5,6,7,8,1)
myfunction(x)
```

Functions

- Example: Consider a random sample $x_1, x_2, \ldots, x_n$ with a sample mean $\bar{x}$ and sample variance s. The $100(1-\alpha)\%$ t-confidence interval for the mean is

$$\bar{x} \pm t_{1-\alpha/2} \times \frac{s}{\sqrt{n}},$$

  where $t_{1-\alpha/2}$ is the critical value for the t distribution with $(n-1)$ degrees of freedom.
- If $x_1, x_2, \ldots, x_n$ is a random sample from a normal population then the t-confidence interval of the mean is exact and is approximately correct for large n otherwise.
- So how large of an n is necessary?
- To investigate this question, we will write a R function that performs a simulation for a given statistical distribution and a sample size.

Functions

```
sim.t.CI <- function(RVgen,n,nsim=1000,alpha=.05,mu,...){
  # Inputs:
  # RVgen: Random variable generator, i.e. rnorm
  # n: Sample size, possibly a vector
  # nsim=1000, number of iterations, default 1000
  # alpha=.05, 100(1-alpha) confidence level
  # mu: Population mean
  # ... Arguments to be passed to RVgen
  CP <- array(dim=nsim)
       for(j in 1:nsim) {
       # Generate random sample
       x <- RVgen(n, ...)
       # Calculate t-confidence interval
       xbar <- mean(x)
       lower <- xbar- qt(1-alpha/2, n-1)*sqrt(var(x)/n)
       upper <- xbar+ qt(1-alpha/2, n-1)*sqrt(var(x)/n)
       CP[j] <- ifelse(lower < mu & mu < upper, 1, 0)
       }
       result=mean(CP)
  return(result)
}
```

Functions

```
> sim.t.CI(rnorm, n=30, mu=0, mean=0,sd=1)
[1] 0.952
> sim.t.CI(rnorm, n=100, mu=0, mean=0,sd=1)
[1] 0.952

> sim.t.CI(rchisq, n=30, mu=3, df=3)
[1] 0.936
> sim.t.CI(rchisq, n=100, mu=3, df=3)
[1] 0.945

> sim.t.CI(runif, n=5, mu=.5, min=0, max=1)
[1] 0.933
> sim.t.CI(runif, n=100, mu=.5, min=0, max=1)
[1] 0.958
```