

# Introduction to R

Stefan Bekiranov

# R

- An integrated suite of tools for data analysis and graphical display.
- R:
  - is an extensive and coherent collection of tools for statistics and data analysis.
  - is an interactive computing environment.
  - provides graphical facilities for data analysis and display.
  - An object-oriented programming language that is *constantly* being extended by the user community.

# Language layout

- Commands are either
  - Expressions (evaluated and printed) or  
`> 2*pi`  
`[1] 6.283185`
  - Assignments (evaluated and pass the value to a variable)  
`> a = 2*pi`  
`> a <- 2*pi`
- Commands are separated by either a semi-colon ; or a newline.
- The # symbol marks the rest of the line as a comment.
- The prompt is > unless the command is syntactically incomplete, when the prompt changes to +.
- Multiple assignments are evaluated from right to left  
`> b <- a <- 6 # 6 is first passed to a and then to b.`
- Several commands can be grouped together as an expression by placing them within braces { ... }
  - Value of grouped expression is value of last command.
  - Used to specify functions.

# Vectors and Matrices

- Most used objects of R users are vectors, matrices, lists and functions
- Six basic types of vectors: logical, integer, double, complex, character
  - Types cannot be mixed
- Different ways to create a vector:

```
> w <- c(1,2,9.8,7,5,pi) # concatenate the values
```

```
> w # all values coerced doubles
```

```
[1] 1.000000 2.000000 9.800000 7.000000 5.000000 3.141593
```

```
> v <- 1:10 # integers from 1 to 10
```

```
> v
```

```
[1] 1 2 3 4 5 6 7 8 9 10
```

```
> colors <- c("red", "green", "blue", "white") # vector of character strings; can use '...' or "...".
```

```
> colors # will always print "..."
```

```
[1] "red" "green" "blue" "white"
```

```
> v[4] # 4th element of vector v
```

```
[1] 4
```

```
> colors[4] # 4th element of vector colors
```

```
[1] "white"
```

```
> v > 6 # what will this do?
```

# Vectors and Matrices (cont'd)

- Different ways to extract subsets of values from a vector:

```
> v > 6 # creates logical vector
```

```
[1] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
```

```
> letters #letters is built in character vector of the 26 letters of alphabet
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
> a <- letters[1:10] # can use vectors of integers to extract a subset of values
```

```
> a
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
> a[v > 6] # can use logical conditions in vector to extract values! This is a very useful feature of R.
```

```
[1] "g" "h" "i" "j"
```

```
> v[v > 6]
```

```
[1] 7 8 9 10
```

```
> a[c(7,8,9,10)]
```

```
[1] "g" "h" "i" "j"
```

```
> names(v) <- letters[1:10] # can name the elements of a vector
```

```
> v
```

```
a b c d e f g h i j
```

```
1 2 3 4 5 6 7 8 9 10
```

```
> v["g"] # can now access the 7th value of v by using its name
```

```
g
```

```
7
```

```
> names(v) = NULL # remove the names
```

```
> a[-c(2,4,6,8,10)] # values can be omitted from a vector by using negative indices
```

```
[1] "a" "c" "e" "g" "i"
```

# Vectors and Matrices (cont'd)

- Create a matrix from a vector using `matrix()` function:

```
> m <- matrix(1:10, nrow=2, byrow=TRUE) # specified number of rows and filled by row
```

```
> m
```

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

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

```
[2,]  6  7  8  9 10
```

```
> ma <- matrix(a, ncol=2) # specified the number of columns and filled by column
```

```
> ma
```

```
      [,1] [,2]
```

```
[1,] "a" "f"
```

```
[2,] "b" "g"
```

```
[3,] "c" "h"
```

```
[4,] "d" "i"
```

```
[5,] "e" "j"
```

- Accessing specific values of a matrix:

```
> m[2,3]
```

```
[1] 8
```

```
> ma[2,2]
```

```
[1] "g"
```

# Vectors and Matrices (cont'd)

- Different ways to extract subsets of values from a matrix:

```
> m > 3 # creates a logical matrix
```

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

```
[1,] FALSE FALSE FALSE TRUE TRUE
```

```
[2,] TRUE TRUE TRUE TRUE TRUE
```

```
> m[m > 3] # creates a vector of values
```

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

```
> m[2,2:4] # vector from 2nd row and columns 2 to 4.
```

```
[1] 7 8 9
```

```
> m[2,] # vector from 2nd row
```

```
[1] 6 7 8 9 10
```

```
> m[,5] # vector from 5th column
```

```
[1] 5 10
```

```
> m[,2:4] # matrix comprised all rows and columns 2 to 4
```

```
  [,1] [,2] [,3]
```

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

```
[2,]  7  8  9
```

```
> m[1,]>3 # logical vector based on 1st row values > 3
```

```
[1] FALSE FALSE FALSE TRUE TRUE
```

```
> m[,m[1,]>3] # subset columns based on 1st row values > 3
```

```
  [,1] [,2]
```

```
[1,]  4  5
```

```
[2,]  9 10
```

# Lists

- List collects together items of different types.

```
> Empl <- list(employee=c("Fred", "Barney"), spouse=c("Wilma", "Betty"), age=c(35,32), wage=c(10.5,8.7))
```

```
> Empl # Components and elements of the list are displayed
```

```
$employee # Component
```

```
[1] "Fred" "Barney" # Elements
```

```
$spouse
```

```
[1] "Wilma" "Betty"
```

```
$age
```

```
[1] 35 32
```

```
$wage
```

```
[1] 10.5 8.7
```

- Accessing elements of a list using \$ or [[index]]:

```
> Empl$spouse # Can access elements of 2nd component by name
```

```
[1] "Wilma" "Betty"
```

```
> Empl$spouse[1] # Element 1 of 2nd component can be accessed using index
```

```
[1] "Wilma"
```

```
> Empl[[2]] # Can access elements of 2nd component by index
```

```
[1] "Wilma" "Betty"
```

```
> Empl[[2]][2] # 2nd element of 2nd component can be accessed using index
```

```
[1] "Betty"
```



# Lists (cont'd)

```
> Empl[3:4] # Access multiple components using [3:5].
```

```
$age
```

```
[1] 35 32
```

```
$wage
```

```
[1] 10.5 8.7
```

```
> Empl[[3:4]] # Don't use [[3:4]] to try to access multiple components!
```

```
Error in Empl[[3:4]] : recursive indexing failed at level 2
```

```
> Empl <- c(Empl, new = "Jetson") # Add new elements using c(); Elements don't have to be the same length.
```

```
> unlist(Empl) # Function unlist converts a list to a vector with names.
```

```
employee1 employee2 spouse1 spouse2 age1 age2 wage1 wage2 new
```

```
"Fred" "Barney" "Wilma" "Betty" "35" "32" "10.5" "8.7" "Jetson" # Character string vector
```

```
> unlist(Empl, use.names=F) # We can remove names using use.names=F option.
```

```
[1] "Fred" "Barney" "Wilma" "Betty" "35" "32" "10.5" "8.7" "Jetson"
```

```
> fourcomp <- c(list(x=1:3, a=3:5), list(y=7:8, b=c(3, 39))) # one list with four components
```

```
> onecomp <- c(list(x=1:3, a=3:5), list(y=7:8, b=c(3, 39)), recursive=T) # if recursive = T, list arguments are  
unlisted before being joined
```

```
> onecomp # vector of length 10
```

```
x1 x2 x3 a1 a2 a3 y1 y2 b1 b2
```

```
1 2 3 3 4 5 7 8 3 39
```

# Factors and Data Frames

- Factor is a special type of vector that holds categorical variable.
  - Gene expression data from wild type (wt), mutant (mt) and mutants treated with TNF $\alpha$  (mut-tnf)
  - Factor with three levels corresponding to three treatments: wt, mut, mut-tnf

```
> exp <- factor(c("wt", "wt", "wt", "mut", "mut", "mut", "mut-tnf", "mut-tnf", "mut-tnf"))
```

```
> Exp # three replicates per treatment
```

```
[1] wt  wt  wt  mut  mut  mut  mut-tnf mut-tnf mut-tnf
```

```
Levels: mut mut-tnf wt
```

- Data frame is a matrix-like object whose columns may be of differing type (i.e., numerical, logical, character).

```
> GeneExp <- c(2.1, 5.5, 14.2) # Numerical vector
```

```
> RespTNF <- c(F, T, T) # Logical vector
```

```
> GeneName <- c("p53", "NFkB", "cMyc") # Character string vector
```

```
> AffyId <- c("affyid1", "affyid2", "affyid3") # Character string vector
```

```
> ExpData <- data.frame(GeneExp, RespTNF, GeneName, row.names=AffyId, stringsAsFactors=F)
```

```
> ExpData # Must set stringsAsFactors variable False or will convert all characters vectors to factors
```

	GeneExp	RespTNF	GeneName
affyid1	2.1	FALSE	p53
affyid2	5.5	TRUE	NFkB
affyid3	14.2	TRUE	cMyc

# Data Frames (cont'd)

- Accessing subsets of values from data frames:

```
> ExpData[1,] # 1st row, like a matrix
```

```
GeneExp RespTNF GeneName  
affyid1  2.1  FALSE   p53
```

```
> ExpData[,2] # 2nd column, like a matrix
```

```
[1] FALSE TRUE TRUE
```

```
> ExpData$GeneExp # 1st column name, like a list
```

```
[1] 2.1 5.5 14.2
```

```
> row.names(ExpData) # creates a vector of row names
```

```
[1] "affyid1" "affyid2" "affyid3"
```

```
> ExpData[row.names(ExpData)=="affyid2",] # use a logical condition, like a matrix
```

```
GeneExp RespTNF GeneName  
affyid2  5.5  TRUE   NFkB
```

```
> ExpData[ExpData$RespTNF,] # use the logical vector in the data frame
```

```
GeneExp RespTNF GeneName  
affyid2  5.5  TRUE   NFkB  
affyid3 14.2  TRUE   cMyc
```

# Object Attributes

- All objects have two attributes: mode and length

```
> mode(v)
```

```
[1] "numeric"
```

```
> length(v) # Number of elements in vector
```

```
[1] 10
```

```
> mode(ma)
```

```
[1] "character"
```

```
> length(ma) # Number of elements in matrix
```

```
[1] 10
```

```
> mode(EmpI) # Mode is list because of mixed types
```

```
[1] "list"
```

```
> length(EmpI) # Number of components in list
```

```
[1] 4
```

```
> mode(ExpData) # Data frame is a list because of mixed types
```

```
[1] "list"
```

```
> length(ExpData) # Number of columns (like components of a list) in data frame
```

```
[1] 3
```

```
> mode(RespTNF)
```

```
[1] "logical"
```

```
> length(RespTNF)
```

```
[1] 3
```

# Object Attributes (cont'd)

- Can access the names of vector elements, lists, and data frame columns using `names()`:

```
> names(Empl)
```

```
[1] "employee" "spouse"   "age"      "wage"
```

```
> names(ExpData)
```

```
[1] "GeneExp" "RespTNF" "GeneName"
```

```
> names(ma)
```

```
NULL
```

- Can access the dimensions of a matrix or data frame using `dim()`:

```
> dim(ma)
```

```
[1] 5 2
```

```
> dim(ExpData)
```

```
[1] 3 3
```

```
> dim(v)
```

```
NULL
```

```
> dim(Empl)
```

```
NULL
```

```
> nrow(ma)
```

```
[1] 5
```

```
> ncol(ma)
```

```
[1] 2
```

# Calling Conventions for Functions

- Functions may have arguments specified or unspecified when the function is defined
  - There may be an arbitrary number of unspecified arguments
  - Unspecified arguments denoted by ...
  - Specified arguments may be supplied in the same order in which they occurred in the function definition
  - Specified arguments may be supplied as name=value in which case their order is not important

> `help(t.test)` # if you know the name of the R built in function, you can use `help()` to get usage information

```
t.test(x, y = NULL, alternative = c("two.sided", "less", "greater"), mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95, ...)
```

> `x <- rnorm(10)` # 10 numbers randomly drawn from a normal distribution;  $x \sim N(0, 1)$

> `y <- rnorm(10)` # 10 numbers randomly drawn from a normal distribution;  $y \sim N(0, 1)$

> `t.test(x, y, "greater")` # arguments in same order in which they are defined in function

> `t.test(x=x, alternative="greater", y=y)` # argument names specified but in wrong order

Welch Two Sample t-test

data: x and y

t = 1.1862, df = 16.896, p-value = 0.1260

alternative hypothesis: true difference in means is greater than 0

95 percent confidence interval:

-0.2838161      Inf

sample estimates:

mean of x   mean of y

0.02149336 -0.58618035

> `t.test(x, "greater", y)` # argument names not specified and in wrong order

Error in `match.arg(alternative)` :

'arg' must be NULL or a character vector

# R Functions and Packages

- The R Base Package (so many functions; indexed by alphabet!)
  - <http://stat.ethz.ch/R-manual/R-patched/library/base/html/00Index.html>
  - Basic functions that come with your installation of R
  - `mean()`; `sum()`; `median()`; `quantile()`; `max()`; `min()`; `range()`;
  - `abs()`; `sign()`; `log()`; `log10()`; `sqrt()`; `exp()`; `sin()`; `cos()`; `tan()`; `sinh()`; `tanh()`
  - `sort()`; `order()`; `rev()`;
  - `duplicated()`; `unique()`;
  - `seq()`; `rep()`;
  - `round()`; `trunc()`; `floor()`; `ceiling()`
  - `cat()`; `paste()`; `substring()`; `grep()`
  - `merge()`; `cbind()`; `rbind()`
- Contributed Packages: Currently, the CRAN package repository features 15466 packages:
  - <http://cran.r-project.org/web/packages/>
  - Specialized packages implementing the latest methods developed in computational statistics.
- Use `help()` for assistance on usage!

# Element wise and Scalar Operators and their Precedence

- `$` (list extraction)
- `[ [` (vector and list element extraction)
- `^ **` (exponentiation)

> m^2

```
[,1] [,2] [,3] [,4] [,5]  
[1,]  1   4   9  16  25  
[2,] 36  49  64  81 100
```

- `-` (unary minus; negative values)
- `:` (sequence generation)
- `%% %/% %*%` (special/matrix operators)
- `* /` (multiply, divide)

> m/m

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

- `+ -` (addition, subtraction)

> m+m

```
[,1] [,2] [,3] [,4] [,5]  
[1,]  2   4   6   8  10  
[2,] 12  14  16  18  20
```

- `< > <= >= == !=` (comparison operators)
- `!` (logical negation)
- `& | && ||` (element wise and scalar logical operators)
- `~` (formula)
- `<- -> =` (assignment)



# Reading Data Into R

- Set working directory

```
> setwd("~/courses/BIOC_8145/intro_R/example")
```

```
> list.files()
```

```
[1] "data_frame.txt" "matrix.txt"    "vector.txt"
```

- Read In Data

- Reading in tables of mixed types: numeric, character and logical

```
> df <- read.table("data_frame.txt", header=T, as.is=T, sep="\t") # as data.frame
```

```
> df
```

```
  Data Class Logic
```

```
1  7.5  Plant TRUE
```

```
2  6.0  Animal FALSE
```

```
3 100.0 Microbe TRUE
```

- Reading in a vector

```
> v <- scan("vector.txt") # reading in a vector
```

```
Read 3 items
```

```
> v
```

```
[1] 3.0 2.2 7.0
```

# Reading Data into R (cont'd)

- Reading in a matrix

```
> m <- scan("matrix.txt")
```

Read 9 items

```
> m # Not what I want!
```

```
[1] 3.0 9.1 1.7 2.2 4.0 11.0 7.0 3.2 7.8
```

- Let's try read.table()

```
> m <- read.table("matrix.txt", header=F, sep="\t")
```

```
> m # This will work, but I get row indexes and column names. What about a simple matrix?
```

```
  V1 V2 V3
```

```
1 3.0 9.1 1.7
```

```
2 2.2 4.0 11.0
```

```
3 7.0 3.2 7.8
```

```
> m <- matrix(scan("matrix.txt"), byrow=T, ncol=3)
```

Read 9 items

```
> m # A simple numerical matrix.
```

```
  [,1] [,2] [,3]
```

```
[1,] 3.0 9.1 1.7
```

```
[2,] 2.2 4.0 11.0
```

```
[3,] 7.0 3.2 7.8
```

# Writing Data to a File

- `write()` writes vector or matrix to a file

> `t(m)` # transpose of matrix m: rows -> columns and columns -> rows

```
 [,1] [,2]
```

```
[1,]  1  6
```

```
[2,]  2  7
```

```
[3,]  3  8
```

```
[4,]  4  9
```

```
[5,]  5 10
```

> `tm <- t(m)`

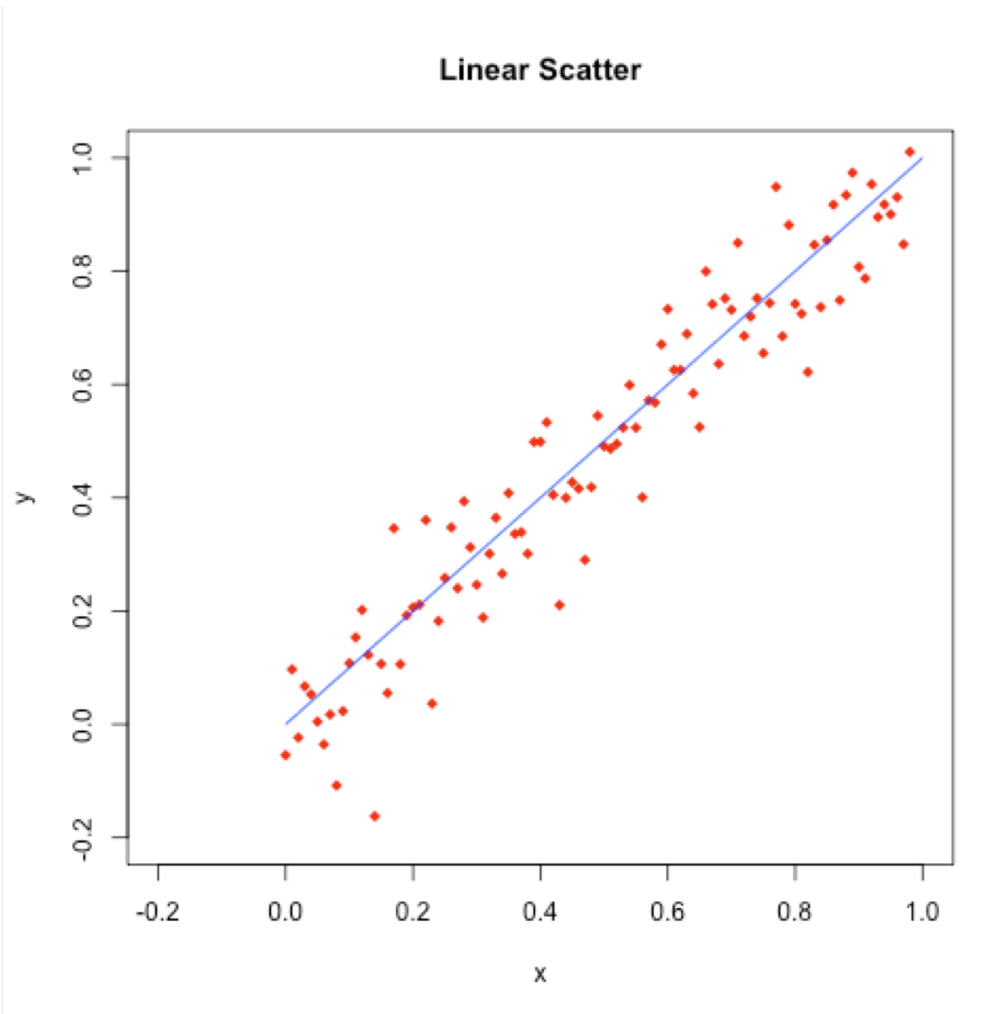
> `write(tm, file="matrix.txt", ncol=5, sep="\t")` # must transpose matrix m to preserve order of values in file

- `write.table()` writes a data frame to a file

> `write.table(ExpData, file="ExpData.txt", quote=F, sep="\t", row.names=T, col.names=T)` # `row.names=T` and `col.names=T` are the default; it's useful to show you they can be omitted.

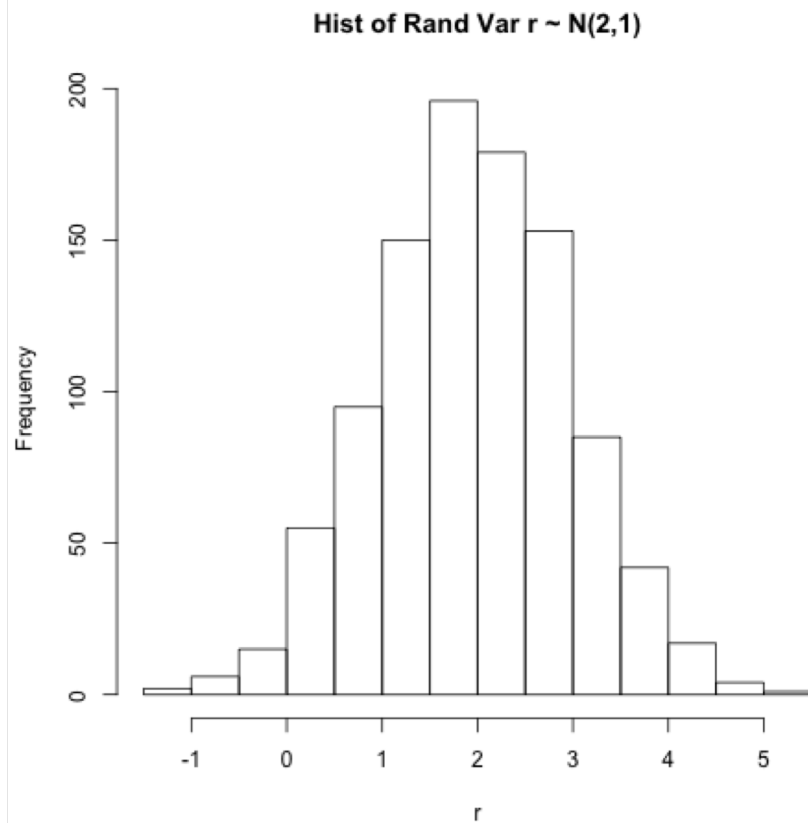
# Graphics

```
> x <- seq(0,1,by=0.01)
> y <- x + rnorm(length(x), mean=0, sd=0.1)
> png(filename="linear_scatter.png")
> plot(x, y, xlab="x", ylab="y", main="Linear Scatter", xlim=c(-.2,1), ylim=c(-.2,1), pch=18, col="red")
> lines(x,x,col="blue")
> graphics.off()
```

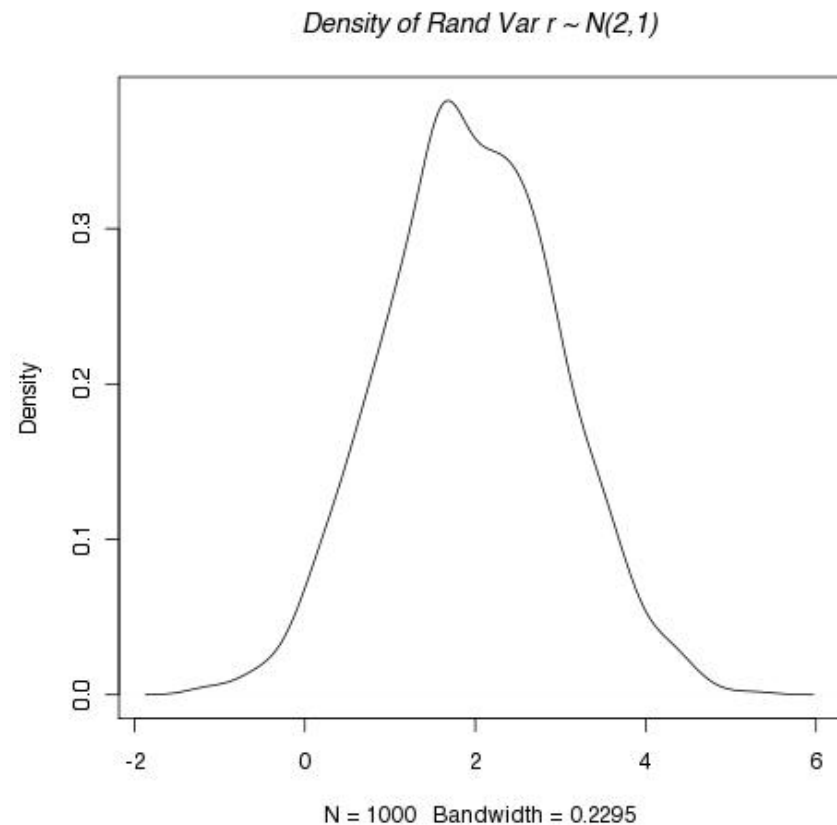


# Graphics (cont'd)

```
> bmp(filename="hist.bmp")  
> hist(r, main="Hist of Rand Var r ~ N(2,1)")  
> graphics.off()
```

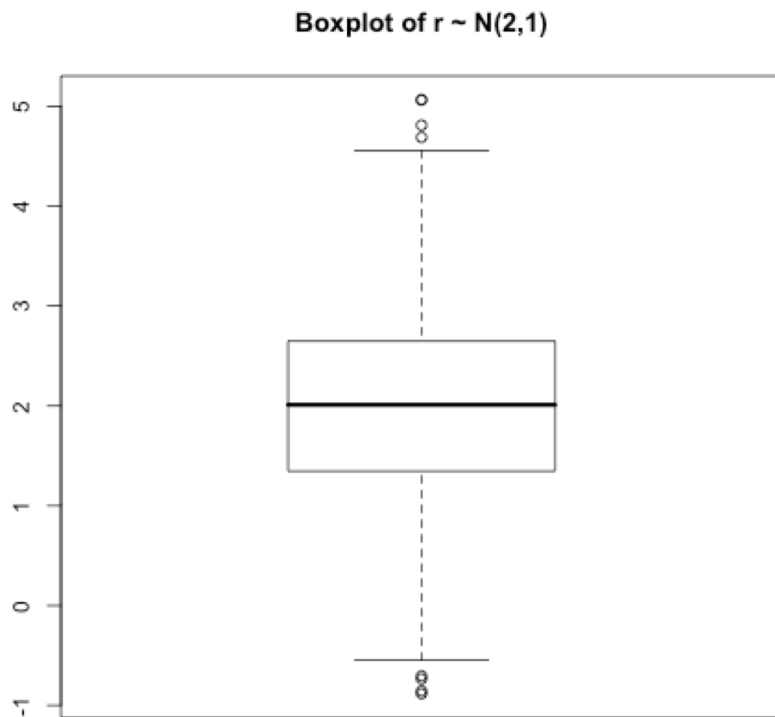


```
> jpeg(filename="density.jpeg")  
> plot(density(r), main="Density of Rand Var r ~ N(2,1)")  
> graphics.off()
```

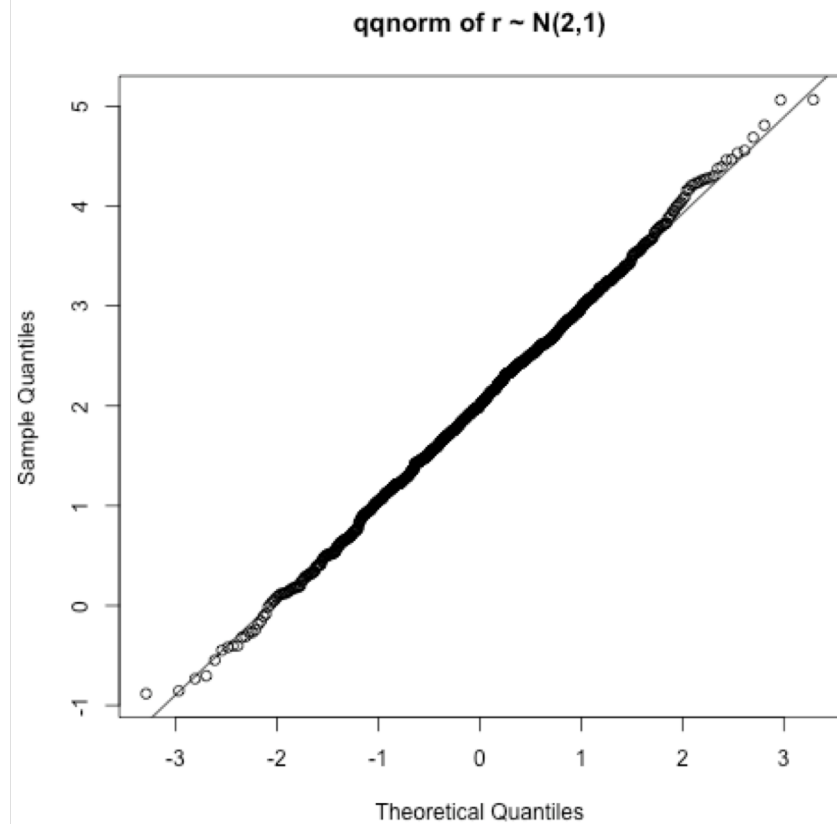


# Graphics (cont'd)

```
> png("boxplot.png")  
> boxplot(r, main="Boxplot of r ~ N(2,1)")  
> graphics.off()
```



```
> png("qqplot.png")  
> qqnorm(r, main="qqnorm of r ~ N(2,1)")  
> qqline(r)  
> graphics.off()
```



# Control Structures

- `if (condition) true.branch else false.branch`

```
> if (length(v) > 10) {  
+ long <- TRUE  
+ variance <- var(v)  
+ } else {  
+ long <- FALSE  
+ variance <- NA  
+ }  
> long  
[1] FALSE  
> variance  
[1] NA
```

- `for (variable in sequence) statement`

— Poor performance (memory and speed!) for large sequences!

```
> square_root <- numeric()  
> for (i in 1:length(v)) {  
+ square_root <- c(square_root, sqrt(v[i]))  
+ }  
> square_root  
[1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000 3.162278
```

- `while (condition) statement`

```
> x<-1; i<-1; sum_sqrt<-0; while (square_root[i] <= 2) {sum_sqrt <- sum_sqrt + square_root[i]; i<-i+1}  
> sum_sqrt  
[1] 6.146264
```

# User Defined Functions

- `function.name <- function(x, y, z, ...) { statement }`

```
> med_mean <- function(x) {if (length(x) > 10) { mean(x) } else { median(x) }}
```

```
> data <- c(rnorm(6), 10) # one outlier among 6 data points drawn from a N(0,1)
```

```
> med_mean(data) # mean would have been badly skewed
```

```
[1] 0.8537254
```

```
> data <- c(rnorm(1000), 10) # one outlier among 1000 points drawn from a N(0,1)
```

```
> med_mean(data) # outlier does not severely impact estimate
```

```
[1] 0.06433641
```

```
> gt1 <- function(x) (sum(x>1)) # counts the number of elements that are greater than 1
```

```
> for (i in 1:100) {data = rbind(data, rnorm(10))} # creates a 200 x 10 random matrix
```

```
> dim(data)
```

```
[1] 200 10
```

```
> rows <- apply(data, 1, gt1) # apply the gt1 function over rows of data; treating each row as a vector
```

```
> columns <- apply(data, 2, gt1) # apply gt1 over columns of data; treating each column as a vector
```

```
> length(rows) # rows should be a vector of length 200
```

```
[1] 200
```

```
> length(columns) # columns should be a vector of length 10
```

```
[1] 10
```

```
> columns
```

```
[1] 42 32 29 42 30 30 33 22 24 27
```



# Working in R

- Use manuals, tutorials, books and web pages (that I provided and others you can find) to familiarize yourself with built in functions.
- Use `help()` function for usage
  - Complete but not anything like a tutorial
  - Written by and for statisticians and knowledgeable users of R
  - Helpful examples of usage listed at bottom of page (cut and paste into session)
- Common workflow: output of function1 -> input to function2 ...
  - Issues: output of function1 incompatible with input requirements of function 2
    - function1 outputs a data frame and function2 only takes a matrix as input
  - Get comfortable with all the major R objects: lists, vectors, matrices, data frames, and functions
  - Coercion: series of functions as.xxx:
    - `as.matrix`: numerical data frame to numerical matrix
    - `as.vector`: attempts to coerce an object into a vector of specified length and mode
    - `as.character`: attempts to coerce an object into a character type
    - `unlist`: converts a list to a vector of its elements
- Avoid for loops over large sequences!