



# R Programming

## -defining functions and programs in R

Martin He

[martin.he@utoronto.ca](mailto:martin.he@utoronto.ca)

# Outline

- Preliminaries
- Basics of R
- Defining functions and programs
- R and bioinformatics

# What is R?

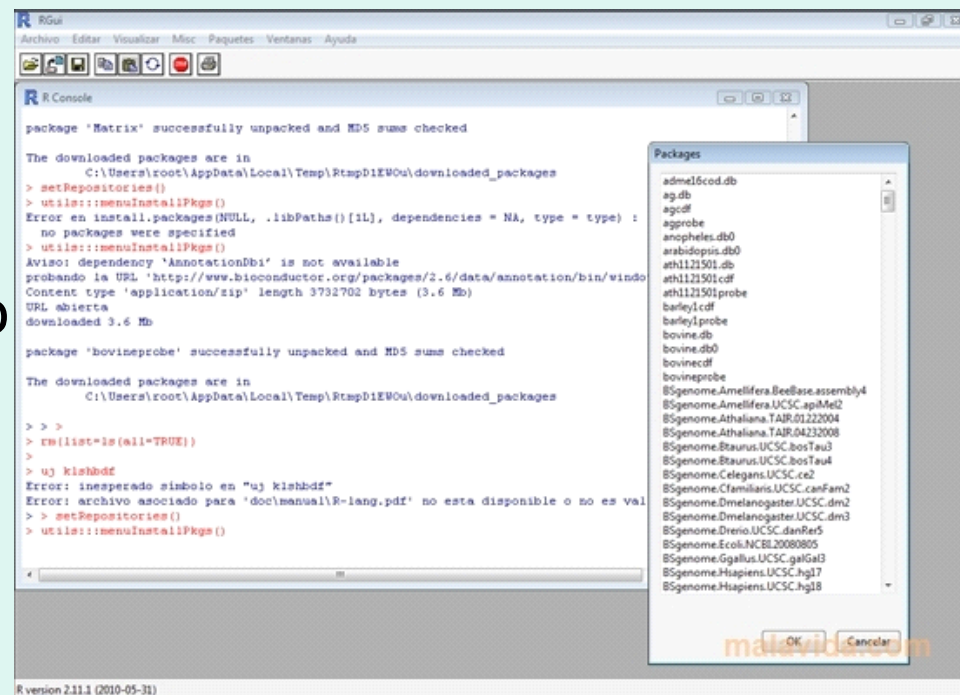
- R is a programming environment for statistics and graphics
  - Data handling(input,output)
  - Matrix operation
  - Statistical tests
  - Graphics
  - Highly specialized data analysis
- Originally developed by Robert Gentleman and Ross Ihaka as the open-source version of the S programming language by John Chambers

# R:core and packages

- R core
  - language interpreter
  - User interface (GUI)
  - Graphics terminal
  - Suite of essential tools for statistics and graphics
- Contributed packages
  - Specialized data analysis(e.g. microarrays) or graphics
  - Any researcher can develop and submit packages
  - Bioconductor is a project for the development of genomic data analysis packages

# Preliminaries

- Installing R
  - Go to
  - <http://cran.r-project.org/>
  - select the latest and appropriate version to install on your machine
  - Install and open, it should look like this



The screenshot shows the R GUI interface. The R Console window displays the following text:

```
package 'Matrix' successfully unpacked and MD5 sums checked
The downloaded packages are in
  C:\Users\root\AppData\Local\Temp\RtmpD1EW0u\downloaded_packages
> setRepositories()
> utils::menuInstallPkgs()
Error en install.packages(NULL, .libPaths()[1L], dependencies = NA, type = type) :
no packages were specified
> utils::menuInstallPkgs()
Aviso: dependency 'AnnotationDbi' is not available
probando la URL 'http://www.bioconductor.org/packages/2.6/data/annotation/bin/windows
Content type 'application/zip' length 3732702 bytes (3.6 Mb)
URL abierta
downloaded 3.6 Mb
package 'bovineprobe' successfully unpacked and MD5 sums checked
The downloaded packages are in
  C:\Users\root\AppData\Local\Temp\RtmpD1EW0u\downloaded_packages
>>
> rm(list=ls(all=TRUE))
> uj klahbdf
Error: inesperado simbolo en "uj klahbdf"
Error: archivo asociado para 'doc/manual/R-lang.pdf' no esta disponible o no es val
>> setRepositories()
> utils::menuInstallPkgs()
```

The Packages dialog box is open, showing a list of available packages:

- adme16cod.db
- ag.db
- agcdf
- agprobe
- anopheles.db0
- arabidopsis.db0
- ath1121501.db
- ath1121501.cdf
- ath1121501probe
- barley1cdf
- barley1probe
- bovine.db
- bovine.db0
- bovinecdf
- bovineprobe
- B5genome.Amelifera.BeeBase.assembly4
- B5genome.Amelifera.UCSC.aplMe2
- B5genome.Athaliana.TAIR.01222004
- B5genome.Athaliana.TAIR.04222008
- B5genome.Etaurus.UCSC.bosTau3
- B5genome.Etaurus.UCSC.bosTau4
- B5genome.Celegans.UCSC.ce2
- B5genome.Cfamilaris.UCSC.canFam2
- B5genome.Dmelanogaster.UCSC.dm2
- B5genome.Dmelanogaster.UCSC.dm3
- B5genome.Dreio.UCSC.danRef5
- B5genome.Ecoli.NCBI.20080805
- B5genome.Ggallus.UCSC.galGa3
- B5genome.Hsapiens.UCSC.hg17
- B5genome.Hsapiens.UCSC.hg18

At the bottom of the dialog box, there are "OK" and "Cancelar" buttons. A watermark "malevids.com" is visible in the bottom right corner of the dialog box.

R version 2.11.1 (2010-05-31)

# Preliminaries

- To use R as an calculator, type 2+5 and hit ENTER(But note how R prints the result)
- To create variables in R,use either -> or = to assign values to variables

```
# approach 1  
>a = 5  
>a  
[1] 5  
# approach 2  
>a = 5 ; a  
# approach 3  
> b <- 5 ; b
```

# Vectors

- A vector is an object composed of an ordered collection of elements of the same type
- Vectors have a mode(or type) and a length
  - the basic modes are : logical , numeric , complex and character

```
#creating a vector
```

```
> x = c(1,2,3,4);
```

```
#length
```

```
>length (x)
```

```
[1] 4
```

```
#mode
```

```
>mode(x)
```

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

# Vector indexes

- To access a subset of the vector, use indexes:the first element is associated to index1,etc
- The attempt to extract an element that does not exist will produce an error
- However,you will be able to assign a value to a position that does not exist yet
- Vector element can also be accessed
  - Using textual label associated to elements
  - Using vectors of logical values (only elements with a corresponding true value will be extracted)



# Vectors

- For vectors with equal spacing , using seq():

```
#create a vector from 1 to 3 with 0.5 increments
```

```
> e = seq(from = 1 , to = 3 , by = 0.5) ; e
```

```
[1] 1.0 1.5 2.0 2.5 3.0
```

- For Vectors of a given length, use rep():

```
> f = rep (na , 6) ; f
```

```
[1] NA NA NA NA NA NA
```

# Lists

- Lists provide a way of storing objects of different types in a single container
  - it has a length and has mode "list"
  - the length of a list is the number of objects it was created from, not the total num of elems

```
>x = list (17 , "A" , TRUE);
```

```
>length(x)
```

```
[1] 3
```

```
>mode(x)
```

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

```
>y = list ( c(1,2,3) , c(4,5))
```

```
>length(x)          #What's the length of list y
```

```
[1] 2
```

# Matrices

- Matrices and arrays can be regarded as the 2-dimensional extensions of vectors
- To create a matrix, use the `matrix()` function:

```
>mat <- matrix (1:6 ,nrow =3, ncol = 2) ; mat
```

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

**#What if we want a 3x2 matrix that take 1->6 row-wise**

```
>mat <- matrix (1:6 ,nrow =3, ncol = 2,byrow = T) ; mat
```

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

# Matrices

- In analogy to vectors, there are different ways to access the matrix elements
  - Numerical indexes
  - Logical vaues
  - Text labels(rownames, colnames)

# Matrices

- To find a tranpose of a matrix, use t():

```
>t (mat)
  [,1] [,2] [,3]
[1,]  1  3  5
[2,]  2  4  6
```

- Similarly, use function dim() to find the dimension of a matrix, use nrow() and ncol() to find the number of rows and columns

# Arithmetic operations of matrices

- Matrix and scalar:
  - every element of the matrix is operated, using the scalar
  - Addition, subtraction, multiplication, division....
- Matrix and vector:
  - The vector is treated as a matrix with only one row or one column
  - with recycling if required
- Matrix and matrix:
  - Element by element (compatible dimensions required)
  - Matrix product(similar to dot product)

# Data types

- Beside numbers and strings, there are some other data types in R which are very useful in statistical data analysis
- Factors
- Data frames

# Factors

- A good deal of statistical data is of a form which indicates which one of several possible categories that an observation falls into, like gender: female, male
- We can use `factor()` and `ordered()`
- The function `factor()` creates data objects which represent variables containing unordered categorical data.



# Factors

```
> eyes = c("blue","hazel","brown","green","brown","blue","blue","brown")
> eyecol = factor(eyes)
> eyes
[1] "blue" "hazel" "brown" "green" "brown" "blue" "blue" "brown"

> eyecol
[1] blue  hazel brown green brown blue  blue  brown
Levels: blue brown green hazel
#What's the difference here
```

- Elements of the set of possible categories that a categorical variable can take on are known as levels of that factor , it is printed when a factor is printed, but can also be retrived with the level() function

# Data frames

- A data frame is similar to a matrix but every column can have different type (numeric, character, logical, factor)
- It provide a way of grouoing a number of vectors and/or factors and return a single object containing the variables.
- Because data frames have a simple retangular row/column layout, it is tempting to treat them as matrices, but this is conceptually wrong and can lead to very inefficient computations.

# Data frames

- An example

```
#create a simple dataframe of gender&hight data set
#use data.frame() to create data frames
> gender = factor(rep(c("female","male") , c(4,4)))
> height = c(165,176,171,177,176,193,180,193)
> classinfo = data.frame(data.frame(gender,height))
> classinfo
  gender height
1 female   165
2 female   176
3 female   171
4 female   177
5  male   176
6  male   193
7  male   180
8  male   193
```

# Data frames

- The underlying representation of data frames is as a named list of vectors and factors, so we can extract elements by name

```
>classinfo$height
[1] 165 176 171 177 176 193 180 193

#To get the average height for male and female
>tapply(classinfo$height , classinfo$gender , mean)
female  male
172.25 185.50
```

# Workspace

- the workspace is the collection of all the objects you have created in a specific R session
  - To list all objects in your workspace  
`ls ( )`
  - To remove object(s) from the workspace  
`rm ( objects )`
  - to remove all objects in the workspace  
`rm ( list = ls ( ) )`

# Workspace

- You can save all objects in your workspace, for use in another session, wither using the GUI or use commands
- use `save (objects , file = filename)` or `save.image (file = filename)` if you want to save all objects

# Working directory

- Mind that the workspace will be saved to a file located in the current working directory
- To change the working directory use the GUI or the following command

```
# use setwd(path)  
setwd ("C:\Users\Yourname\Documents\Data")
```

- To check what's in the current working directory

```
getwd()
```

# Importing datasets to R

- To import data sets that are not an R data set object(i.e. do not have a .RData extension):
  - check what folder R is working with right now:
  - tell R what folder the dataset is stored(if different from the current directory)
  - use the `read.table()` command to read in the data
- For data sets that are an R data set object(i.e. have .RData extension)
  - load the data into R from the console  
`load("dataSetName.RData")`
  - or simply double click on the file



# R graphics

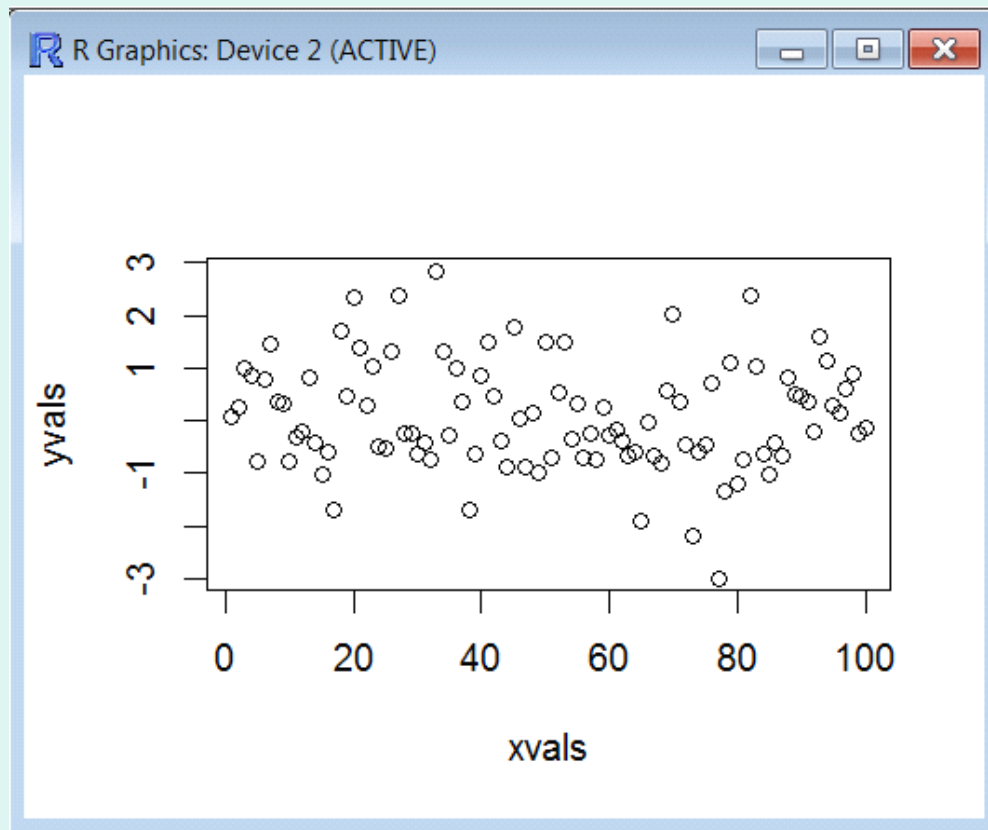
- R has a number of functions that produce graphics on screen.
- When these functions are called, a graphics window is opened on screen (if one is not already open) and the graphs are drawn or rendered there.
- Some function draw an entire graph with a single function call, they are called high-level graphics functions.
- Other graphics functions are used to build up graphs incrementally, they are called low-level graphics function

# The plot functions

- The most basic plot function in R is called plot.
- It takes a set of x and y coordinates and produces a plot based on these coordinates
- The simplest plot is scatterplot, but optional arguments made it possible to produce a variety of different plot styles
- Other optional arguments control features such as axis labelling and annotation

# An example

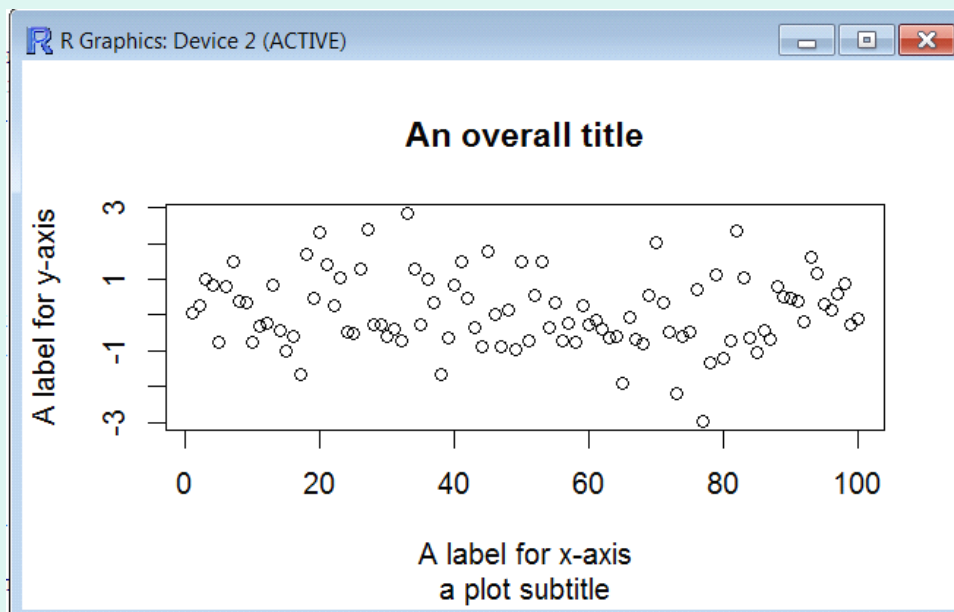
```
> xval = 1:100  
> yval = rnorm(100)  
> plot ( xvals , yvals)
```



# Customising labelling

- The default annotation and labelling can be overridden with optimal `main,xlab,ylab` and `sub` arguments

```
>plot(xvals , yvals , main = "An overall title" , xlab = "A label for x-axis" ,  
      ylab = "A label for y-axis" , sub = "a plot subtitle")
```



# Defining functions and programs in R

- Built-in functions in R
- Define simple functions
- Flow of control
- Evaluation
- More complicated functions

# Functions

- All computation in R are carried out by calling functions
- a call to an R function takes zero or more arguments and return a single variable
- Defining function provides user a way of adding new functionality to R
- function define by users have the same status as the function built into R

# Built-in function

- Here is a collection of some of the most basic and useful built-in function in R

## 1.Numeric function

Function	Description
abs (x)	absolute value
sqrt (x)	square root
ceiling (x)	ceiling of (3.45) is 4
floor (x)	floor of ( 3.45) is 3
trunc (x)	trunc (5.999) is 5
round (x , digits = n)	round a number to desired digits
signif (x , digits = n )	round number to specific number of significant digits
log (x)	natural logarithm
exp (x)	$e^x$

# Built-in function

## 2.statistical function

Function	Description
mean (x)	mean of object x
sd (x)	standard deviation of object x
median (x)	median
quantile (x , probs)	quantiles where x is the numeric vector whose quantile are desired
range (x)	range
sum (x)	sum
diff ( x, lag = 1 )	lagged difference
min (x) , max (x)	maximum ; minimum
scale (x , center = TRUE, scale = TRUE)	column center or standardize a matrix.



# First step: define a simple function

- We define a function which returns the square of its argument
  - we can use this function in exactly the same way as any other R function
  - Because the `*` operator act elements-wise on vectors, the square function we created will act the same way

```
> square = function (x) x * x
```

```
>square (10)
```

```
[1] 100
```

```
>square (1:5)
```

```
[1] 1 4 9 16 25
```

# Functions defined in terms of other functions

- The square function is no different from any other R function and we can use it in other function definitions
  - e.g. we can define a function that returns the sum of the squares based on the square function we just defined

```
> sumsq = function (x) sum (square (x) )
```

```
>sumsq (1:10)
```

```
[1] 385
```

# Example:the which function

- This is a very useful R function, called **which** that takes a logical vector and returns the indexes of the values in the vector that are true,here is an example:

```
#runif() function can generate a set of random uniform values
> u = runif (5)
> u
[1] 0.1408437 0.3629749 0.2368168 0.5566332 0.2924938

>which (u > 0.5)
[1] 4
which (0.25 < u & u < 0.75)
[1] 2 4 5
```

# Exercise: define **which** function

- Some Tips

```
#We can get a set of logical values stored in a vector x
```

```
> x = u > 0.5
```

```
>x
```

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

```
#We can obtain the indexes of elements of the vector using this expression
```

```
> 1:length(x)
```

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

```
#We can retrieve the indexes corresponding to the TRUE elements of x by using logical subsetting
```

```
> a = c(1, 2, 3, 4, 5) ; a
```

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

```
> b= c(TRUE,FALSE,FALSE,FALSE,TRUE); b
```

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

```
> a [b]
```

```
[1] 1 5
```

# Exercise: define **which** function

- We can now create a simple version of the which function by defining a suitable function

```
#define our own which function  
>myWhich = function (x) (1:length (x) ) [x]
```

- THE FUNCTION WORK AS EXPECTED.

```
>myWhich (u > 0.5)  
[1] 4  
>myWhich(0.25 < u & u < 0.75)  
[1] 2 4 5
```

# Some issues

- The expression `1:length(x)` works fine if `x` has non-zero length
- It does not do what is required when `x` has length 0

```
>x = logical (0) ; x
logical(0)
>length(x)
[1] 0
>1:length (x)
[1] 1 0
```

# Some issues

- To avoid this problem it is best to use the alternative expression `seq(along = x)` to generate the indexes of the values in `x`.

```
#How seq(along = x) behaves
> x = (u > 0.5) ; x
[1] FALSE FALSE FALSE TRUE FALSE
> seq(along = x)
[1] 1 2 3 4 5
#The same as before

#Here is the difference
> 1:length(x)
[1] 1 0
> seq(along = x)
integer(0)
```

# Some Issues

- The myWhich function we have defined assumed that its argument are logical and it can produce strange answers if it is not.
- It also doesn't act like the system function when there are NA values present

```
> x = c(0,1,0,1)
```

```
#The predefined which function can recognize this issue
```

```
> which(x)
```

```
Error in which(x) : argument to 'which' is not logical
```

```
#But the function we defined can not recognize this and produce results
```

```
#that doesn't make much sense
```

```
> myWhich(x)
```

```
[1] 1 1
```

```
>x = c(TRUE , NA , FALSE)> which(x)
```

```
[1] 1
```

```
> myWhich(x)
```

```
[1] 1 NA
```



# Some issues

- We can improve our which function as follows.

```
> myWhich = function (x) +  
+ seq(along = x) [!is.na(as.logical(x)) & as.logical(x)]
```

```
#Here is what as.logical do
```

```
>as.logical ( c(TRUE , FALSE , TRUE) )
```

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

```
> as.logical(c(TRUE,FALSE,NA))
```

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

```
> as.logical(c(TRUE,FALSE,0,1))
```

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

```
#Here is what !is.na(as.logical(x)) do
```

```
> !is.na(as.logical(c(TRUE,FALSE,FALSE)))
```

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

```
> !is.na(as.logical(c(TRUE,FALSE,FALSE,NA)))
```

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

```
> !is.na(as.logical(c(TRUE,FALSE,FALSE,0,1)))
```

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

# Now we are good

- Repeating tests shows all the issues are fixed

```
#Recognizes 0 and 1
```

```
> myWhich (c ( 0 , 1 , 0 , 1))
```

```
[1] 2 4
```

```
#Recognizes NA
```

```
> myWhich (c ( TRUE , NA ,FALSE))
```

```
[1] 1
```

```
#Works with 0 length vector
```

```
> myWhich (c (logical(0)))
```

```
integer(0)
```

# More complicated functions

- Functions can be required to carry out some very complex calculation
- Such calculations can require more than a simple expression used in the which function
- In order to create complicated functions we need to know more about R languages and about how to direct flow of control

# Expression and compound expressions

- R programs are made up of expressions. These can either be simple expressions or compound expression consisting of simple expressions separated by semicolons or newlines and grouped within braces.

{ expr1 ; expr2 ; ..... ; exprn }

- Every expression in R has a value the the value of the compound expression above is the value of exprn, e.g.

```
>x = { 10 , 20 }
```

```
>x
```

```
[1] 20
```

# Assignments within compound expressions

- It is possible to have assignments within compound expression and the values of the variables and the values of variables which this procedure can be used in later expressions

```
>z = { x = 10 , y = x^2 , x+y}
```

```
>x
```

```
[1] 10
```

```
>y
```

```
[1] 100
```

```
>z
```

```
[1] 100
```

# If-then-else statements

- If-then-else statements make it possible to choose between two( possibly compound ) expressions depending on the value of the condition

`if ( condition ) expr1 else expr2`

- If condition is true then expr1 is evaluated otherwise expr2 is executed

## Note

- only the first element in condition is checked.
- The value of the whole expression is the value of whichever expression was executed

# If-then-else examples

- The expression

`if (x > 0) y = sqrt (x) else y = -sqrt(-x)`

provides an example of an if-then-else statement which will look familiar to Java, C, or Python programs we have seen

- This statement can, however, be written more succinctly in R as

`y = if (x > 0) sqrt (x) else -sqrt (-x)`

which will look similar to functional languages like Lisp or Haskell

# If-then-else statements

- There is a simplified form of if-then-else statement which is available when there is no *expression2* to evaluate

*if (condition) expression*

and this is completely equivalent to the statement

*if (condition) expression else NULL*



# Combining Logical Conditions

- The & and | operators works element-wise on vector
- When programming, it is only the first element of the logical vector which is important
- There are special logical operators && and || which work on just the first elements of their argument

```
> c(TRUE , FALSE ) & TRUE  
[1] TRUE FALSE  
> c(TRUE ,FALSE ) && TRUE  
[1] TRUE
```

# Combining Logical conditions

- The `&&` and `||` operator also evaluate just enough of their argument as is necessary to determine their result

```
> TRUE && print (TRUE)
[1] TRUE
[1] TRUE
> TRUE || print (TRUE)
[1] TRUE
```

- In the first case both arguments are evaluated and in the second case only the first argument is evaluated because the entire statement is known to be true when the first argument is seen to be TRUE

# For loops

- As part of a computing task we often want to repeatedly carry out some computation for each element of a vector or list. In R this is done with a for loop.
- A for loop has the form:  
`for(variable in vector) expression`
- The effect of such a loop is to set the value of variable equal to each element of the vector in turn, each time evaluating the given expression.

# For loop example

- Suppose we have a vector  $x$  which contains a set of numerical values, and we want to compute the sum of those values. One way to carry out the calculation is to initialize a variable to zero and to add each element in turn to that variable.

```
> s = 0
> for (i in 1 : length ( x )
      s = s + x [i]
```

- The effect of this calculation is to successively set the variable  $i$  equal to each of the values  $1, 2, \dots, \text{length}(x)$ , and for each of the successive values to evaluate the expression  $s = s + x[i]$ .

# For loop Example 2

- The previous example is typical of loop in many computer programming languages , but R does not need to use an integer loop variable
- the loop could instead be written

```
>s = 0  
>for(elt in x)  
  s = s + elt
```

- This is both simpler and more efficient

# The "next" statement

- Sometimes, when given conditions are met, it is useful to skip to the end of the loop, without carrying out the intervening statements. This can be done by executing a next statement when the conditions are met.

```
for(variable in vector) {  
    expression1  
    expression2  
    if (condition)  
        next  
    expression3  
    expression4  
}
```

- When condition is true ,expression3 and expresison4 are skipped

# A small exercise

- Use the for loop and "next" statement to sum just the positive elements of a vector  $x$
- use the style that are more favorable in R

# A possible solution

```
s = 0
for (elem in x) {
  if (elem <= 0)
    next
  s = s + elem
}
```

#It does this by skipping to the end of the loop when it encounters non-positive values



# The "break" statement

- The break statement is similar to the next statement but, rather than jumping to the end of the loop, it stops the execution of the loop and jumps to the following statement

```
for(variable in vector) {  
    expression1  
    expression2  
    if (condition)  
        break  
    expression3  
    expression4  
}
```

# While loops

- For loops evaluate an expression a fixed number of times, while loops repeat until a particular condition is false
- A while-loop looks like:  
`while (condition) expression`
- Again, condition is an expression which must evaluate to a simple logical value, and expression is a simple or compound expression

# A simple exercise

- Suppose we have dividend = 22 and divisor = 5, show how to carry out a long division using while loops
- The result of 22 divided by 5 is 4 with a remainder of 2

# A possible solution

```
> dividend = 22
> divisor = 5
> wholes = 0
> remainder = dividend
> while (remainder > divisor) {
  remainder = remainder - divisor
  wholes = wholes + 1
}>c (wholes , remainder)
[1] 4 2
```

# A square root algorithm

- There is a very famous method for computing square roots devised by Isaac Newton.
- The method works by taking advantage of the fact that if  $g$  is a guess at the square root of  $x$  then an improved guess can be obtained by taking the average of  $g$  and  $x/g$ .
- If  $g$  is bigger than the square root of  $x$  then  $x/g$  will be smaller and conversely.
- By averaging the bigger and smaller values we get a value which is closer to the square root of  $x$  than either of them.
- Starting with a guess of 1, we keep improving the guess until it provides a good enough approximation.

# Computing square roots

- This small piece of code shows how the computation proceeds, roughly speaking, the number of correct digits doubles every iteration

```
> guess = numeric ( 5 )
> x = 2
> g = 1
> for (i in 1:5) {
    g = 0.5 * (g + x/g)
    guesses [ i ] = g
}
>guesses
[1] 1.500000 1.416667 1.414216 1.414214
[5] 1.414214
```

# A square root function

- We can put the previous algorithm/code into a function that computes square root

```
>root =  
  function (x) {  
    g=1  
    for (i in 1:5)  
      g = 0.5 * (g + x / g)  
    g  
  }
```

```
>root (2)  
[1] 1.414214
```

# Returning values

- So far, our functions have used the structure of the code to decide which values will be returned ((the value is that of the last expression in the function body)
- It is possible to return a value from anywhere inside a function using a call to the return function.
- This looks like `return (value)`
- Calls to return function should be used sparingly (if at all) because they make it harder to understand the structure of code.
- In essence, a call to return amounts to doing a `goto`, which is generally frowned upon in programming.



# Functions in general

- In general, an R function definition has the form:  
function (arglist) body
- where:  
**arglist** is a (comma separated) list of variable names known as the formal arguments of the function, **body** is a simple or compound expression known as the body of the function.
- Functions are usually, but not always, assigned a name so that they can be used in later expressions.

# Evaluation of Functions

- Function evaluation takes place as follows:
  - (i) Temporarily create a set of variables by associating the arguments passed to the function with the variable names in arglist.
  - (ii) Use these variable definitions to evaluate the function body.
  - (iii) Remove the temporary variable definitions.
  - (iv) Return the computed values.

# Evaluation example

- Evaluating the function call

`hypot (3, 4)`

takes places as follows:

(i) Temporarily create variables `a` and `b`, which have the values 3 and 4.

(ii) Use these values to compute the value (5) of `sqrt(a^2 + b^2)`.

(iii) Remove the temporarily variable definitions

(iv) Return the value 5

# Optional Arguments

- It is possible to declare default values for arguments so that specifying values for those arguments is optional.
- The default values are specified by following the argument by an = sign followed by an expression which defines the value.
- The expression defining default values can include variables which are arguments to the function or which are defined in the body of the function.

# An example

- The following function computes the sum of squares(a staple quantity in statistical analysis).

```
> sumsq =  
  function (x, about = mean (x))  
    sum ( (x - about ) ^2 )
```

- By default the function computes the sum of squared deviations around the sample mean, but the optional second argument makes other possibilities available.

```
>sumsq (1:10)  
[1] 82.5
```

```
>sumsq(1:10 , 0)  
[1] 385
```

# Lazy evaluation

- The expression given as default values of arguments are not evaluated until they are required.
- This is referred to as lazy evaluation.
- In the example:

```
> sumsq =  
  function(x , about = mean( x ) )  
  {  
    x = x [!is.na (x)]  
    sum( (x - about ) ^ 2 )  
  }
```

- The default value for about is not computed until it is needed to evaluate the expression  $\text{sum}((x - \text{about})^2)$

# Vectorization

- In general, R functions defined on scalar values are expected to operate element-wise on vectors.
- The standard mathematical and string handling functions all obey this general rule.
- Often this kind of vectorization happens naturally as a side effect of the way R operates.

# Invisible return values

- All R functions return a value. It is possible to make the value returned by a function be non-printing by returning it as the value of the invisible function

```
>no.print =  
  function(x)  
    invisible (x^2)  
>no.print (1:3)  
#nothing printed on screen  
  
>x = no.print(1:3)  
>x  
[1] 1 4 9
```



# Variable Numbers of Arguments

- R functions can be defined to take a variable number of arguments. The special argument name ... will match any number of arguments in the call (which are not matched by any other arguments)
- The mean function computes the mean of values in a single vector. We can easily create an equivalent function which will compute the mean of all the values in all its arguments

```
>myMean =  
  function(...)  
  mean(c(...))
```

```
>myMean (1:3 , 1:5)  
[1] 2.625
```

# Some restrictions on ...

- only one ... can be used in the formal argument list for a function
- the only thing which can be done with ... inside a function is to pass it as an argument to a function call
- Argument which follow ... in a formal argument list must be specified by name, and the name cannot be abbreviated

# Using ...

- The following function assembles its arguments (twice) into a vector.

```
> c2 = function (...)  
  c( ... , ...)
```

```
> c2 (1, 2 , 3 )  
[1] 1 2 3 1 2 3
```

- Notice that argument name are passed along with ...

```
> c2 ( a = 1 , b = 2 )  
a b a b  
1 2 1 2
```

# An example

- Sometimes a situation arises during a computation where it is necessary to simply give up and abandon the computation.
- There is a R function called `stop` which makes this easy
- The argument to `stop` is a character string which explains why the computation is being stopped

```
> fake.fun = function (x) if (x>10) stop ("bad x value") else x
```

```
> y = fake.fun(5) ; y  
[1] 5
```

```
> y = fake.fun(15)  
Error in fake.fun(15) : bad x value
```

# Warnings

- Occasionally, situations arise where it could be that something has gone wrong, but it isn't completely clear that it has.
- In such a situation it can be useful to continue the computation, but to also alert the user that there is a potential problem.
- The warning function can be used to issue warnings in these cases, e.g.

```
> if ( any ( wrts < 0 ) )  
    warning ("negative weights encountered")
```

# General advice

- Write your own functions and (re)use them everywhere.
- Writing functions will make you a better programmer.
- If you find your self copy and pasting blocks of code, write a function instead.
- Read other people's function/code (R makes this tremendously easy).
- Do not reinvent the wheel, particularly if that the wheel is in base.

# Need more help?

- The above discussion of programming in R is not conclusive there you are sure to encounter situations that you need some further knowledge of programming in R

# R help

- For help with any functions in R, put a question mark before the function name to determine what arguments to use, examples and background information.

- For example:

`?plot`



The screenshot shows a web browser window displaying the R Documentation page for the 'help' function. The browser's address bar shows the URL '127.0.0.1:16680/library/utils/html/help.html'. The page content includes a title 'help {utils}', a 'Description' section stating 'help is the primary interface to the help systems.', a 'Usage' section with the command: `help(topic, package = NULL, lib.loc = NULL, verbose = getOption("verbose"), try.all.packages = getOption("help.try.all.packages"), help_type = getOption("help_type"))`, and an 'Arguments' section listing parameters: 'topic' (usually a name or character string), 'package' (a name or character vector), 'lib.loc' (a character vector of directory names), 'verbose' (logical), 'try.all.packages' (logical), and 'help\_type' (character string).



# Some online resources for R

- R project home: <http://www.r-project.org>
- Search Engine for R: <http://rseek.org>
- R Reference Card:<http://cran.r-project.org/doc/contrib/Short-refcard.pdf>
- R Graph Gallery: <http://addictedtor.free.fr/graphiques/>
- Statistics with R: <http://zoonek2.free.fr/UNIX/48R/all.html>
- Springer (useR! series):<http://www.springerlink.com/home/main.mpx>
- A introduction to R: <http://cran.r-project.org/doc/manuals/R-intro.pdf>

# R and bioinformatics

- Bioconductor
  - Introduction
  - use of Bioconductor
  - Installation
  - Resources

# Bioconductor



Search:

[Home](#)

[Install](#)

[Help](#)

[Developers](#)

[About](#)

## About Bioconductor

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data.

Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, [516 software packages](#), and an active user community.

## Use Bioconductor for...

### ➔ [Microarrays](#)

Import Affymetrix, Illumina, Nimblegen, Agilent, and other platforms. Perform quality assessment, normalization, differential expression, clustering, classification, gene set enrichment, genetical genomics and other workflows for expression, exon, copy number, SNP, methylation and other assays. Access GEO, ArrayExpress, Biomart, UCSC, and other community resources.

### ➔ [High Throughput Assays](#)

Import, transform, edit, analyze and visualize flow cytometric, mass spec, HTqPCR, cell-based, and other assays.

### ➔ [Sequence Data](#)

Import fasta, fastq, ELAND, MAQ, BWA, Bowtie, BAM, gff, bed, wig, and other sequence formats. Trim, transform, align, and manipulate sequences. Perform quality assessment, ChIP-seq, differential expression, RNA-seq, and other workflows. Access the Sequence Read Archive.

### ➔ [Annotation](#)

Use microarray probe, gene, pathway, gene ontology, homology and other annotations. Access GO, KEGG, NCBI, Biomart, UCSC, vendor, and other sources.



[Mailing Lists](#)

Subscribe »



[Events](#)



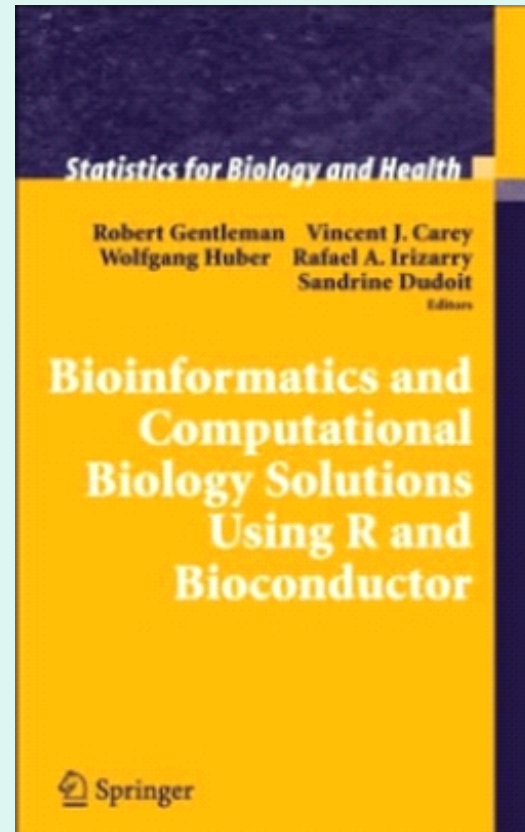
[News](#)

# Bioconductor

- Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data.
  - Microarrays
  - High throughput assays
  - Sequence data
  - Annotation
- Bioconductors uses the R statistical programming language.
  - A collection of R packages
- It is open source and open development
  - It has two releases each year, more than 516 packages, and an active user community.

# Bioconductor

- Begun in 2001, based at Harvard and now FHCRRC(Seattle,WA)
- A large collection of R packages(they also convert to good software to R)



Gentleman et al (above) is a very helpful reference text

# Bioconductor software packages

- Official website: [www.bioconductor.org](http://www.bioconductor.org)
- <http://www.bioconductor.org/packages/release/bioc/>

[Home](#) » [Bioconductor 2.9](#) » 2.9 Software Packages

## Bioconductor Software Packages

Bioconductor version: Release (2.9)

Package	Maintainer	Title
<a href="#">a4</a>	Tobias Verbeke	Automated Affymetrix Array Analysis Umbrella Package
<a href="#">a4Base</a>	Tobias Verbeke	Automated Affymetrix Array Analysis Base Package
<a href="#">a4Classif</a>	Tobias Verbeke	Automated Affymetrix Array Analysis Classification Package
<a href="#">a4Core</a>	Tobias Verbeke	Automated Affymetrix Array Analysis Core Package
<a href="#">a4Preproc</a>	Tobias Verbeke	Automated Affymetrix Array Analysis Preprocessing Package
<a href="#">a4Reporting</a>	Tobias Verbeke	Automated Affymetrix Array Analysis Reporting Package
<a href="#">ABarray</a>	Yongming Andrew Sun	Microarray QA and statistical data analysis for Applied Biosystems Genome Survey Microarray (AB1700) gene expression data.
<a href="#">aCGH</a>	Peter Dimitrov	Classes and functions for Array Comparative Genomic Hybridization data.
<a href="#">ACME</a>	Sean Davis	Algorithms for Calculating Microarray Enrichment (ACME)

# Use of Bioconductor

- Microarray
  - Import Affymetrix, Illumina, Nimblegen, Agilent, and other platforms.
  - Perform quality assessment, normalization, differential expression, clustering, classification, gene set enrichment, genetical genomics and other workflows for expression, exon, copy number, SNP, methylation and other assays.
  - Access GEO, ArrayExpress, Biomart, UCSC, and other community resources.

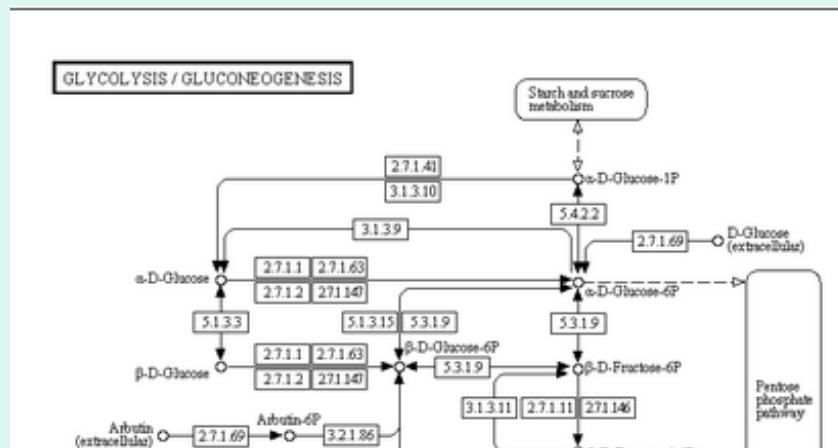
# Use of Bioconductor

- Sequence data
  - Import fasta, fastq, ELAND, MAQ, BWA, Bowtie, BAM, gff, bed, wig and other sequence formats.
  - Trim, transform, align and manipulate sequences.
  - Perform quality assessment, CHIP-seq, differential expression, RNA-seq, and other workflows
  - Access the sequence Read Archive
- High throughput assays
  - Import, transform, edit, analyze and visualize flow cytometric, mass spec, HTqPCR, cell-based, and other assays.



# Use of Bioconductor

- Annotation
  - Use microarray probe, gene, pathway, gene ontology, homology and other annotations.
  - Access GO,KEGG,NCBI, Biomart, UCSC, vendor, and other sources.



<http://www.genome.jp/kegg/>

# Getting started



Home

Install

Help

[Home](#) » [Install](#)

- [Getting The Latest Version of Bioconductor](#) • [Install Packages](#) • [Find Packages](#) • [Update Packages](#)
- [Install R](#)

## Getting The Latest Version of Bioconductor

If you have installed the latest release of R, you will automatically get packages from the latest version of Bioconductor by following the steps below. The current release version of R is 2.14, and the currently released Bioconductor version is 2.9.

## Install Bioconductor Packages

Use the `biocLite.R` script to install Bioconductor packages. To install a particular package, e.g., `limma`, type the following in an R command window:

```
source("http://bioconductor.org/biocLite.R")
biocLite("limma")
```

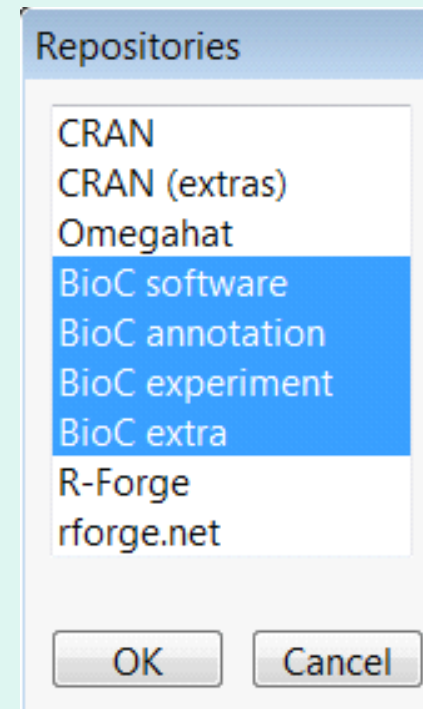
Install several packages, e.g., "GenomicFeatures" and "AnnotationDbi", with

```
biocLite(c("GenomicFeatures", "AnnotationDbi"))
```

<http://www.bioconductor.org/install/>

# Getting started

- Go to
  - <http://www.bioconductor.org/packages/2.9/BiocViews.html>
- Software
- AnnotationData
- ExperimentData
- Extra



# Bioconductor basics

```
> source ("https://bioconductor.org/biocLite.E")  
> biocLite()
```

## •Installs the following libraries:

affy, affydata, affyPLM, annaffy, annotate, Biobase,  
Biostrings, DynDoc, gcrma, genefilter, geneplotter, hgu95av2.db,  
limma, marray, matchprobes, multtest, ROC, vsn, xtable,  
affyQCReport

... then you use e.g. library as before

vignette (package = "ROC") tells you to look  
vignette("ROCnotes") for a worked example - a  
very helpful introduction. (OR use e.g.  
openVignette("ROC") from the Biobase package)

# Bioconductor basics

- To get other packages, use e.g. `biocLite("SNPchip")`
- Do not need to type `biocLite()` after you install (even in a new R session).
- This would install everything again, which is harmless, but slow

# Find help

- Some help functions

```
>library (affy)
```

```
#Loads a particular package (here affy package).
```

```
>library (help=affy)
```

```
#List all functions/objects of a package(here affy package)
```

```
>library()
```

```
#List all libraries/packages that are available on the system
```

```
>openVignette()
```

```
#Provides documentation on packages
```

```
>sessionInfo()
```

```
#Prints version information about R and all loaded packages
```

# Want to learn more?

- The official Bioconductor website:<http://www.bioconductor.org/>
- Bioconductor workshops and courses:<http://www.bioconductor.org/help/course-materials/>
- R programming for bioinformatics by Robert Gentleman
- Bioinformatics and Computational Biology Solutions Using R and Bioconductor by Robert Gentleman et al
- Bioconductor Case Studies by Florian Hahne et al

Thank you...