



# BGGN 213

**More on R functions  
and packages**

**Barry Grant**  
**UC San Diego**

<http://thegrantlab.org/bggn213>

# Recap From Last Time:

- Data frames are created with the `data.frame()` function as well as the `read.table()` family of functions including `read.csv()`, `read.delim()` etc.
- A function is called by giving its name with comma separated input arguments in parentheses e.g. `mean(x, na.rm=TRUE)`
- A function is defined with (1) a user selected **name**, (2) a comma separated set of input **arguments**, and (3) regular R code for the **function body** including an optional output **return value** e.g.
- Rule of thumb: When you find yourself doing the same thing 3 or more times with repetitive code consider writing a function.
- Follow a step-by-step procedure to go from working code snippet to refined and tested function.

- A function is defined with (1) a user selected **name**, (2) a comma separated set of input **arguments**, and (3) regular R code for the **function body** including an optional output **return value** e.g.

```
fname <- function(arg1, arg2) { paste(arg1,arg2) }
```

fname      (arg1, arg2)      { paste(arg1,arg2) }  
Name                  Input arguments          Function body



- A function is defined with **(1)** a user selected **name**, **(2)** a comma separated set of input **arguments**, and **(3)** regular R code for the **function body** including an optional output **return value** e.g.

```
fname <- function(arg1, arg2) { paste(arg1, arg2) }
```

Diagram illustrating the components of an R function definition:

- Name**: `fname`
- Input arguments**: `arg1, arg2`
- Function body**: `{ paste(arg1, arg2) }`

- **(1)** Start with a simple problem and write a working snippet of code. **(2)** Rewrite for clarity and to reduce duplication **(3)** Turn into an initial function **(4)** Test on small well defined input **(5)** Report on potential problem by failing early and loudly!

[MPA Link]

Back by popular demand

**More examples of how to  
write your own functions!**

- (1) Start with a **simple problem** and write a **working snippet** of code. (2) **Rewrite** for clarity and to reduce duplication (3) Turn into an **initial function** (4) **Test** on small well defined input (5) Report on potential problem by failing early and loudly!

# Today's Learning Goals

- **More on data import**

- Pre-check recommendations
- `read.table()` and friends for flat files
- `readxl::read_excel()` for excel files
- RData files

- **Writing your own functions**

- What, Why, When and How

- **The CRAN & Bioconductor R package repositories**

- Rmarkdown, ggplot2, bio3d, rgl and rentrez

- Additional R packages (and development versions of CRAN packages) on **GitHub** and **BitBucket**.

# What is a function

```
1      name.of.function <- function(2      arg1, arg2) {  
      statements  
      3      return(something)  
      }
```

1 **Name** (can be *almost* anything you want )

2 **Arguments** (i.e. input to your function)

3 **Body** (where the work gets done)



# Why would you write a function

When you find yourself doing the same thing 3 or more times it is time to write a function.

```
## What does this code do?
```

```
df$a <- (df$a - min(df$a)) / (max(df$a) - min(df$a))
```

```
df$b <- (df$b - min(df$a)) / (max(df$b) - min(df$b))
```

```
df$c <- (df$c - min(df$c)) / (max(df$c) - min(df$c))
```

```
df$d <- (df$d - min(df$d)) / (max(df$a) - min(df$d))
```

# Why would you write a function

When you find yourself doing the same thing 3 or more times it is time to write a function.

```
## Consider copy and paste errors:
```

```
df$a <- (df$a - min(df$a)) / (max(df$a) - min(df$a))
```

```
df$b <- (df$b - min(df$a)) / (max(df$b) - min(df$b))
```

```
df$c <- (df$c - min(df$c)) / (max(df$c) - min(df$c))
```

```
df$d <- (df$d - min(df$d)) / (max(df$a) - min(df$d))
```

# Why would you write a function

Consider the advantages:

```
## Here the intent is far more clear  
  
df$a <- rescale(df$a)
```

- Makes the purpose of the code more clear
- Reduce mistakes from copy/paste
- Makes updating your code easier
- Reduce duplication and facilitate re-use.

# How would you write this function

Start with a **working code snippet**, simplify, reduce calculation duplication,...

```
## First consider the original code:
```

```
df$a <- (df$a - min(df$a)) / (max(df$a) - min(df$a))
```

```
df$b <- (df$b - min(df$a)) / (max(df$b) - min(df$b))
```

```
df$c <- (df$c - min(df$c)) / (max(df$c) - min(df$c))
```

```
df$d <- (df$d - min(df$d)) / (max(df$a) - min(df$d))
```

# How would you write this function

Start with a working code snippet, **simplify**, reduce calculation duplication,...

```
## Simplify to work with a generic vector named "x"  
x <- (x - min(x)) / (max(x) - min(x))
```

# How would you write this function

Start with a working code snippet, simplify, **reduce calculation duplication**,...

```
## Note that we call the min() function twice..  
x <- (x - min(x)) / (max(x) - min(x))
```



# How would you write this function

Start with a working code snippet, simplify, **reduce calculation duplication**,...

```
## Note that we call the min() function twice..  
  
xmin <- min(x)  
x <- (x - xmin) / (max(x) - xmin)
```

# How would you write this function

Start with a working code snippet, simplify, **reduce calculation duplication**,...

```
## Further optimization to use the range() function..  
  
rng <- range(x)  
x <- (x - rng[1]) / (rng[2] - rng[1])
```

# How would you write this function

Start with a working code snippet, simplify, reduce calculation duplication, finally **turn it into a function**

```
## You need a "name", "arguments" and "body"...

rescale <- function(x) {
  rng <- range(x)
  (x - rng[1]) / (rng[2] - rng[1])
}

# Test on a small example where you know the answer
rescale(1:10)
```

# How would you write this function

Test, **Fail**, Change, Test again,...

```
# Test on a small example where you know the answer  
rescale(1:10)
```

```
# How would you get your function to work here..  
rescale( c(1,2,NA,3,10) )
```

```
# What should your function do here?  
rescale( c(1,10,"string") )
```

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {  
  # Our rescale function from lecture 9  
  
  if(na.rm) {  
    rng <-range(x, na.rm=na.rm)  
  } else {  
    rng <-range(x)  
  }  
  
  answer <- (x - rng[1]) / (rng[2] - rng[1])  
  
  if(plot) {  
    plot(answer, ...)  
  }  
  return(answer)  
}
```

```
source("http://tinyurl.com/rescale-R")
```

# The functions `warning()` and `stop()`

- The functions `warning()` and `stop()` are used inside functions to handle and report on unexpected situations
- They both print a user defined message (which you supply as a character input argument to the `warning()` and `stop()` functions).
- However, `warning()` will keep on going with running the function body code whereas `stop()` will terminate the action of the function.
- A common idiom is to use `stop("some message")` to report on unexpected input type or other problem early in a function, i.e. **fail early and loudly!**



```
rescale2 <- function(x, na.rm=TRUE, plot=FALSE, ...) {  
  if( !is.numeric(x) ) {  
    stop("Input x should be numeric", call.=FALSE)  
  }  
  
  rng <- range(x, na.rm=TRUE)  
  
  answer <- (x - rng[1]) / (rng[2] - rng[1])  
  
  if(plot) {  
    plot(answer, ...)  
  }  
  return(answer)  
}
```

```
source("http://tinyurl.com/rescale-R")
```

```
rescale2 <- function(x, na.rm=TRUE, plot=FALSE, ...) {  
  if( !is.numeric(x) ) {  
    stop("Input x should be numeric", call.=FALSE)  
  }  
  
  rng <- range(x, na.rm=TRUE)  
  
  answer <- (x - rng[1]) / (rng[2] - rng[1])  
  
  if(plot) {  
    plot(answer, ...)  
  }  
  return(answer)  
}
```

```
source("http://tinyurl.com/rescale-R")
```

# Suggested steps for writing your functions

1. Start with a simple problem
2. Get a working snippet of code
3. Rewrite to use temporary variables
4. Rewrite for clarity and to reduce calculation duplication
5. Turn into an initial function
6. Test on small well defined input and (subsets of) real input
7. Report on potential problem by failing early and loudly!

# Side-Note: What makes a good function?

- Correct
- Understandable (remember that functions are for humans and computers)
- Correct + Understandable = **Obviously correct**
- Use sensible names throughout. What does this code do?

```
baz <- foo(df, v=0)  
df2 < replace_missing(df, value=0)
```

- Good names make code understandable with minimal context

# More examples

- We want to write a function, `both_na()` that counts how many positions in two vectors, `x` and `y`, both have a missing value

```
# Should we start like this?  
  
both_na <- function(x, y) {  
  # something goes here?  
}
```

# **No!** Always start with a simple definition of the problem

- We should start by solving a simple example problem first where we know the answer.

```
# Lets define an example x and y
x <- c( 1, 2, NA, 3, NA)
y <- c(NA, 3, NA, 3, 4)
```

- Here the answer should be **1** as only the third position has NA in both **x** and **y** and **is.na()** and **sum()** functions



Get a **working snippet** of code first that is close to what we want

```
# Lets define an example x and y
x <- c( 1, 2, NA, 3, NA)
y <- c(NA, 3, NA, 3, 4)
```

```
# use the is.na() and sum() functions
is.na(x)
[1] FALSE FALSE  TRUE FALSE  TRUE

sum( is.na(x) )
[1] 2

# Putting together!
sum( is.na(x) & is.na(y) )
[1] 1
```

# Then rewrite your snippet as a *first* function

```
# Lets define an example x and y
x <- c( 1, 2, NA, 3, NA)
y <- c(NA, 3, NA, 3, 4)
```

```
# Our working snippet
sum( is.na(x) & is.na(y) )

# No further simplification necessary
both_na <- function(x, y) {
  sum( is.na(x) & is.na(y) )
}
```

# Test on various inputs (a.k.a. **eejit proofing**)

- We have a function that works in at least one situation, but we should probably check it works in others.

```
x <- c(NA, NA, NA)
y1 <- c( 1, NA, NA)
y2 <- c( 1, NA, NA, NA)
```

```
both_na(x, y1)
[1] 2

# What will this return?
both_na(x, y2)
```

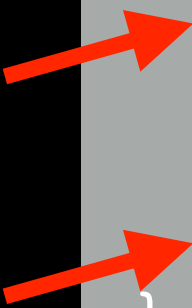
# Report on potential problem by **failing early and loudly!**

- The generic warning with recycling behavior of the last example may not be what you want as it could be easily missed especially in scripts.

```
both_na2 <- function(x, y) {  
  if(length(x) != length(y)) {  
    stop("Input x and y should be the same length", call.=FALSE)  
  }  
  sum( is.na(x) & is.na(y) )  
}
```

# **Refine and polish:** Make our function more useful by returning more information

```
both_na3 <- function(x, y) {  
  
  if(length(x) != length(y)) {  
    stop("Input x and y should be vectors of the same length")  
  }  
  
  na.in.both <- ( is.na(x) & is.na(y) )  
  na.number  <- sum(na.in.both)  
  na.which   <- which(na.in.both)  
  
  message("Found ", na.number, " NA's at position(s):",  
          paste(na.which, collapse=" ", " ") )  
  
  return( list(number=na.number, which=na.which) )  
}
```



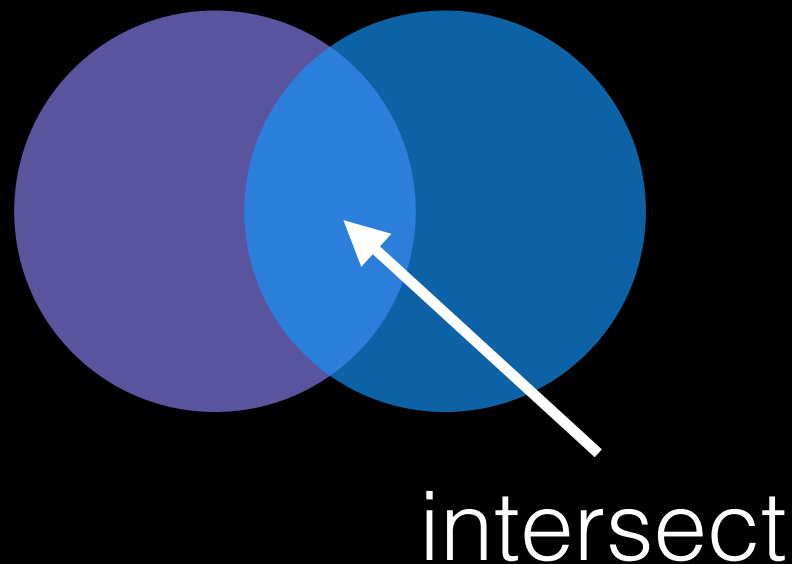
# Re-cap: Steps for function writing

1. Start with a simple problem
2. Get a working snippet of code
3. Rewrite to use temporary variables
4. Rewrite for clarity and to reduce calculation duplication
5. Turn into an initial function
6. Test on small well defined input and (subsets of) real input
7. Report on potential problem by failing early and loudly!
8. Refine and polish
9. Document and comment within the code on your reasoning.



# One last example

Find common genes in two data sets and return their associated data (from each data set)



Follow along!

```
# source("http://tinyurl.com/rescale-R")

# Simplify problem
df1 <- data.frame(IDs=c("gene1", "gene2", "gene3"),
                  exp=c(2,1,1),
                  stringsAsFactors=FALSE)

df2 <- data.frame(IDs=c("gene2", "gene4", "gene3", "gene5"),
                  exp=c(-2, NA, 1, 2),
                  stringsAsFactors=FALSE)

# Simplify further to single vectors
x <- df1$IDs
Y <- df2$IDs

# Now what do we do?
```

```
source("http://tinyurl.com/rescale-R")
```

Follow along!

```
# source("http://tinyurl.com/rescale-R")

# Simplify problem
df1 <- data.frame(IDs=c("gene1", "gene2", "gene3"),
                  exp=c(2,1,1),
                  stringsAsFactors=FALSE)

df2 <- data.frame(IDs=c("gene2", "gene4", "gene3", "gene5"),
                  exp=c(-2, NA, 1, 2),
                  stringsAsFactors=FALSE)

x <- df1$IDs
y <- df2$IDs

# Search for existing functionality to get us started...
??intersect

intersect(x, y)
[1] "gene2" "gene3"
```

```
# Close but not useful for returning indices yet.
```

```
intersect(x, y)
```

```
[1] "gene2" "gene3"
```

```
# Back to the documentation to find something more useful
```

```
??intersect
```

Follow along!

Follow along!

```
# Close but not useful for returning indices yet.
```

```
intersect(x, y)
```

```
[1] "gene2" "gene3"
```

```
# Back to the documentation to find something more useful
```

```
? "%in%"
```

```
# This looks like a more useful starting point - indices!
```

```
x %in% y
```

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

Follow along!

```
# Close but not useful for returning indices yet.
intersect(x, y)
[1] "gene2" "gene3"

# Back to the documentation to find something more useful
?"%in%"

# This looks like a more useful starting point - indices!
x %in% y
[1] FALSE TRUE TRUE

x[x %in% y]
[1] "gene2" "gene3"

y[ y %in% x ]
[1] "gene2" "gene3"

# We can now cbind() these these results to yield intersect
```

Follow along!

```
# Putting together
```

```
cbind( x[ x %in% y ], y[ y %in% x ] )
```

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

```
[1,] "gene2" "gene2"
```

```
[2,] "gene3" "gene3"
```

```
# Make it into a first function
```

Follow along!

```
# Putting together
```

```
cbind( x[ x %in% y ], y[ y %in% x ] )
```

```
      [,1]      [,2]  
[1,] "gene2"  "gene2"  
[2,] "gene3"  "gene3"
```

```
# Make it into a first function
```

```
gene_intersect <- function(x, y) {  
  cbind( x[ x %in% y ], y[ y %in% x ] )  
}
```

```
# Looks good so far but we need to work with data frames
```

```
gene_intersect(x, y)  
      [,1]      [,2]  
[1,] "gene2"  "gene2"  
[2,] "gene3"  "gene3"
```



Follow along!

```
# Previous function for vector input
gene_intersect <- function(x, y) {
  cbind( x[ x %in% y ], y[ y %in% x ] )
}

# Lets change to take input data frames
gene_intersect2 <- function(df1, df2) {
  cbind( df1[ df1$IDs %in% df2$IDs, ],
        df2[ df2$IDs %in% df1$IDs, "exp" ] )
}

# Correct but yucky format for 2nd colnames
gene_intersect2(df1, df2)
  IDs exp df2[df2$IDs %in% df1$IDs, "exp" ]
2 gene2  1                               -2
3 gene3  1                               1
```

Follow along!

```
# Our input $IDs column name may change so lets add flexibility
# By allowing user to specify the gene containing column name

# Experiment first to make sure things are as we expect
gene.colname="IDs"
df1[,gene.colname]
[1] "gene1" "gene2" "gene3"

# Next step: Add df1[,gene.colname] etc to our current function.
```

Follow along!

```
# Looks complicated - simplify for human consumption!
```

```
gene_intersect3 <- function(df1, df2, gene.colname="IDs") {  
  cbind( df1[ df1[,gene.colname] %in% df2[,gene.colname], ],  
        exp2=df2[ df2[,gene.colname] %in% df1[,gene.colname], "exp"] )  
}
```

```
# Works but the function is not kind on the reader
```

```
gene_intersect3(df1, df2)
```

```
  IDs exp exp2  
2 gene2  1  -2  
3 gene3  1   1
```

```
# Looks much better
```

```
gene_intersect4 <- function(df1, df2, gene.colname="IDs") {  
  
  df1.name <- df1[,gene.colname]  
  df2.name <- df2[,gene.colname]  
  
  df1.inds <- df1.name %in% df2.name  
  df2.inds <- df2.name %in% df1.name  
  
  cbind( df1[ df1.inds, ],  
         exp2=df2[ df2.inds, "exp" ] )  
}
```

```
# Getting closer!
```

```
gene_intersect4(df1, df2)  
  IDs exp exp2  
2 gene2  1  -2  
3 gene3  1   1
```

```
# Test, break, fix, text again
```

```
df1 <- data.frame(IDs=c("gene1", "gene2", "gene3"),  
                  exp=c(2,1,1),  
                  stringsAsFactors=FALSE)
```

```
df3 <- data.frame(IDs=c("gene2", "gene2", "gene5", "gene5"),  
                  exp=c(-2, NA, 1, 2),  
                  stringsAsFactors=FALSE)
```

```
# Works but could do with more spit and polish!
```

```
gene_intersect4(df1, df3)
```

```
  IDs exp exp2  
1 gene2  1  -2  
2 gene2  1  NA
```

```
Warning message:
```

```
In data.frame(..., check.names = FALSE) :
```

```
  row names were found from a short variable and have been  
discarded
```

```
# Additional features we could add
# - Catch and stop when user inputs weird things
# - Use different colnames for matching in df1 and df2,
# - Match based on the content of multiple columns,
# - Optionally return rows not in df1 or not in df2 with NAs
# - Optionally sort results by matching column
# - etc...
```

```
merge(df1, df2, by="IDs")
```

	IDs	exp.x	exp.y
1	gene2	1	-2
2	gene3	1	1

Refer to sections in  
last days handout!

<http://tinyurl.com/bgggn213-L9>

Section **Section 1B** has been updated to include  
homework scoring rubric (see question 6).

Other sections are there for your benefit.

R Highlight!

# CRAN & Bioconductor

Major repositories for **R packages**  
that extend R functionality



# **CRAN**: Comprehensive R Archive Network

- CRAN is a network of mirrored servers around the world that administer and distribute R itself, R documentation and **R packages** (basically add on functionality!)
- There are currently ~11,700 packages on CRAN in the areas of finance, bioinformatics, machine learning, high performance computing, multivariate statistics, natural language processing, *etc. etc.*

<https://cran.r-project.org/>

# Side-note: R packages come in all shapes and sizes



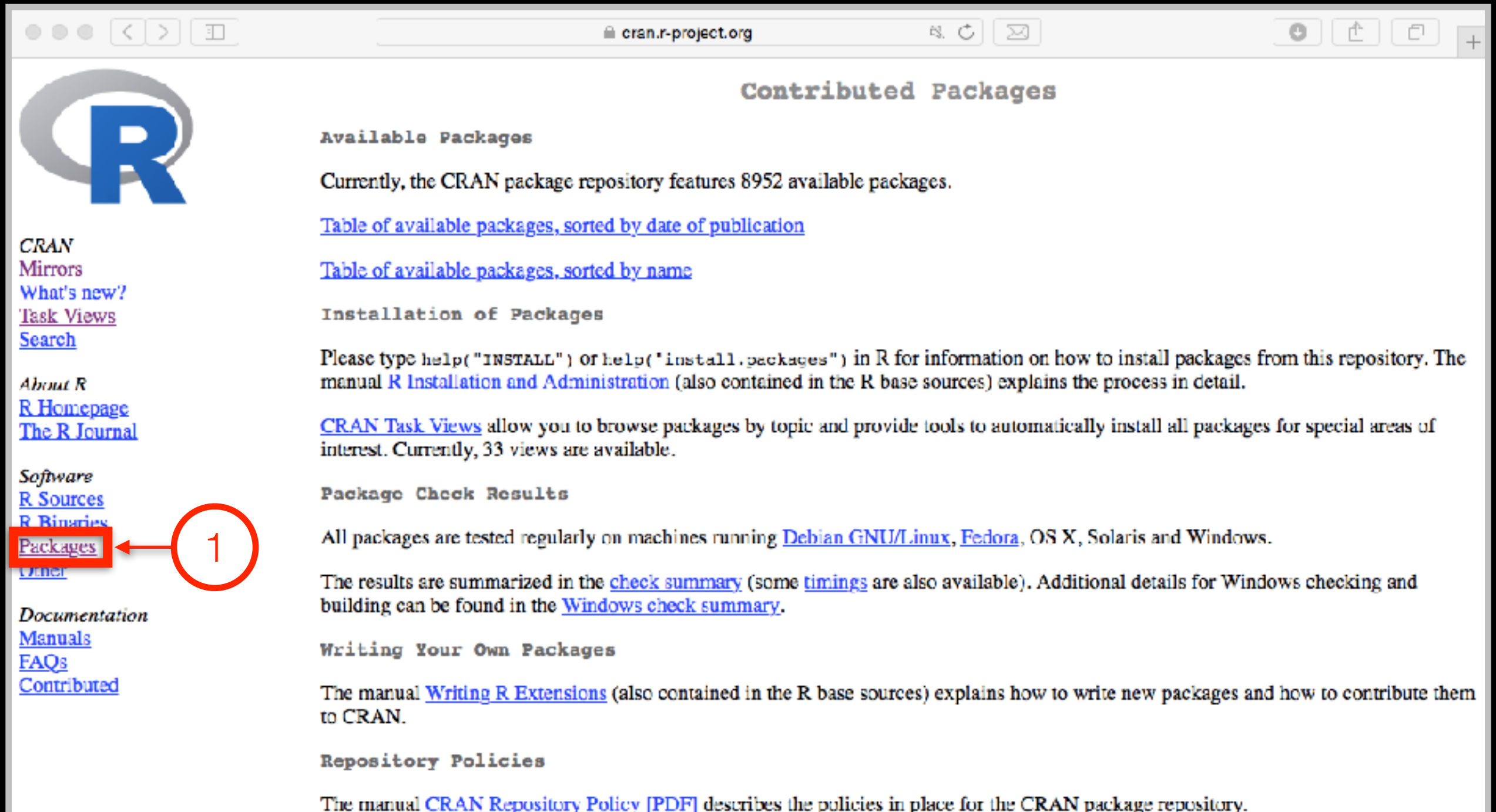
R packages can be of variable quality and often there are multiple packages with overlapping functionality.

**Refer to relevant publications, package citations, update/maintenance history, documentation quality and your own tests!**

“ The journal has sufficient experience with CRAN and Bioconductor resources to endorse their use by authors. We do not yet provide any endorsement for the suitability or usefulness of other solutions. ”

From: “Credit for Code”. *Nature Genetics* (2014), 46:1

# <https://cran.r-project.org>



**Contributed Packages**

**Available Packages**

Currently, the CRAN package repository features 8952 available packages.

[Table of available packages, sorted by date of publication](#)

[Table of available packages, sorted by name](#)

**Installation of Packages**

Please type `help("INSTALL")` or `help("install.packages")` in R for information on how to install packages from this repository. The manual [R Installation and Administration](#) (also contained in the R base sources) explains the process in detail.

[CRAN Task Views](#) allow you to browse packages by topic and provide tools to automatically install all packages for special areas of interest. Currently, 33 views are available.

**Package Check Results**

All packages are tested regularly on machines running [Debian GNU/Linux](#), [Fedora](#), OS X, Solaris and Windows.

The results are summarized in the [check summary](#) (some [timings](#) are also available). Additional details for Windows checking and building can be found in the [Windows check summary](#).

**Writing Your Own Packages**

The manual [Writing R Extensions](#) (also contained in the R base sources) explains how to write new packages and how to contribute them to CRAN.

**Repository Policies**

The manual [CRAN Repository Policy \[PDF\]](#) describes the policies in place for the CRAN package repository.

**CRAN**  
[Mirrors](#)  
[What's new?](#)  
[Task Views](#)  
[Search](#)

*About R*  
[R Homepage](#)  
[The R Journal](#)

*Software*  
[R Sources](#)  
[R Binaries](#)  
**[Packages](#)**  
[Other](#)

*Documentation*  
[Manuals](#)  
[FAQs](#)  
[Contributed](#)

# Installing a package

RStudio > Tools > Install Packages

```
> install.packages("bio3d")
```

```
> library("bio3d")
```

# Your Turn: Pick a package to explore and install

Do it Yourself!

(~~Rmarkdown~~), ggplot2, bio3d, rgl, rentrez, igraph

## Questions to answer:

- How does it extend R functionality? (i.e. What can you do with it that you could not do before?)
- How is it's documentation, vignettes, demos and web presence?
- Can you successfully follow a tutorial or vignette to get started quickly with the package?
- Can you find a GitHub or Bitbucket site for the the package with a regular heartbeat?

[Collaborative Google Doc Link](#)

# Bioconductor

R packages and utilities for working with  
high-throughput genomic data

<http://bioconductor.org>







**More pragmatic:**

Bioconductor is a **software repository of R packages** with **some rules and guiding principles.**

**Version 3.3 had 1211 software packages.**

Bioconductor has  
emphasized

**Reproducible Research**

since its start, and has been  
an early adapter and driver  
of tools to do this.

“Bioconductor: open software development for computational biology and bioinformatics”

Gentleman et al

Genome Biology 2004, 5:R80

“Orchestrating high-throughput genomic analysis with Bioconductor”

Huber et al

Nature Methods 2015, 12:115-121

# Installing a bioconductor package

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

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

# Summary

- R is a powerful data programming language and environment for statistical computing, data analysis and graphics.
- Introduced R syntax and major R data structures
- Demonstrated using R for exploratory data analysis and graphics.
- Exposed you to the why, when, and how of writing your own R functions.
- Introduced CRAN and Bioconductor package repositories.

# Learning Resources

- **TryR**. An excellent interactive online R tutorial for beginners.  
< <http://tryr.codeschool.com/> >
- **RStudio**. A well designed reference card for RStudio.  
< <https://help.github.com/categories/bootcamp/> >
- **DataCamp**. Online tutorials using R in your browser.  
< <https://www.datacamp.com/> >
- **R for Data Science**. A new O'Reilly book that will teach you how to do data science with R, by Garrett Grolemund and Hadley Wickham.  
< <http://r4ds.had.co.nz/> >