Recap From Last Time:

- Covered the When, Why, What and How of writing your own R functions.

...

Recap From Last Time:

- Covered the **When**, Why, What and How of writing your own R functions.

➡ **When**: When you find yourself doing the same thing 3 or more times with repetitive code consider writing a function.

...

Recap From Last Time:

- Covered the When, **Why**, What and How of writing your own R functions.

➡ **When**: When you find yourself doing the same thing 3 or more times with repetitive code consider writing a function.

➡ **Why**:

1. Makes the purpose of the code more clear
2. Reduces mistakes from copy/paste
3. Makes updating your code easer
4. Reduces code duplication and facilitates re-use.

...

Recap From Last Time:

- Covered the When, Why, **What** and How of writing your own R functions.

➡ **What**: 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)  }
```

Name — Input arguments — Function body

. . .

Every function in R has the same parts

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {
   rng <-range(x, na.rm=na.rm)
   answer <- (x - rng[1]) / (rng[2] - rng[1])

   if(plot) {
     plot(answer, ...)
   }

   return(answer)
}
```

Name: We can run the function by typing its name followed by brackets.

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {
   rng <-range(x, na.rm=na.rm)
   answer <- (x - rng[1]) / (rng[2] - rng[1])

   if(plot) {
     plot(answer, ...)
   }

   return(answer)
}
```

Name: We can run the function by typing its name followed by brackets.

Arguments: We can supply values for these variables that appear in the function body.

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {
   rng <-range(x, na.rm=na.rm)
   answer <- (x - rng[1]) / (rng[2] - rng[1])

   if(plot) {
     plot(answer, ...)
   }

   return(answer)
}
```

Name: We can run the function by typing its name followed by brackets.

Arguments: We can supply values for these variables that appear in the function body.

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {
  rng <-range(x, na.rm=na.rm)
  answer <- (x - rng[1]) / (rng[2] - rng[1])
  if(plot) {
    plot(answer, ...)
  }

  return(answer)
}
```

Body: R will run this code whenever we call the function by typing its name followed by brackets.

Name: We can run the function by typing its name followed by brackets.

Arguments: We can supply values for these variables that appear in the function body.

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {
  rng <-range(x, na.rm=na.rm)
  answer <- (x - rng[1]) / (rng[2] - rng[1])
  if(plot) {
    plot(answer, ...)
  }

  return(answer)
}
```

Return value: The function will return this value

Body: R will run this code whenever we call the function by typing its name followed by brackets.

Recap From Last Time:

- **How**: Follow a step-by-step procedure to go from working code snippet to refined and tested function.

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

…

Recap From Last Time:

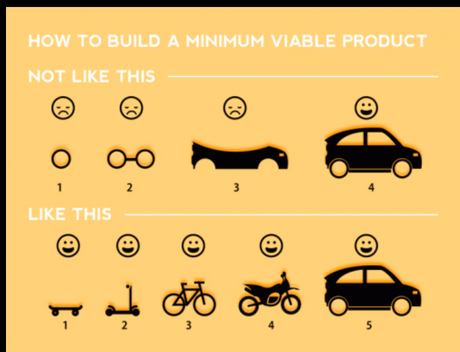
- **How**: Follow a step-by-step procedure to go from working code snippet to refined and tested function.

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

…

Recap…

1. Start with a simple problem and write a working snippet of code.



[Image credit: Spotify development team]

Build that skateboard before you build the car.

A limited but functional thing is very useful and keeps the spirits high.

[MPA link]

Back by popular demand

More examples of how to write your own functions!

Revisit our first example function from last day…

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

```
rescale <- function(x, na.rm=TRUE, plot=FALSE, ...) {
   rng <-range(x, na.rm=na.rm)

   answer <- (x - rng[1]) / (rng[2] - rng[1])

   if(plot) {
     plot(answer, ...)
   }

   return(answer)
}
```

```
# Test fail
rescale( c(1,10,"string") )
```

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!**

```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=na.rm)

   answer <- (x - rng[1]) / (rng[2] - rng[1])

   if(plot) {
      plot(answer, ...)
   }
   return(answer)
}
```

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


```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=na.rm)

   answer <- (x - rng[1]) / (rng[2] - rng[1])

   if(plot) {
      plot(answer, ...)
   }
   return(answer)
}
```

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


Suggested steps for writing your functions

1. Start with a simple problem and get a working snippet of code
2. Rewrite to use temporary variables (e.g. x, y, df, m etc.)
3. Rewrite for clarity and to reduce calculation duplication
4. Turn into an initial function with clear useful names
5. Test on small well defined input and (subsets of) real input
6. Report on potential problem by failing early and loudly!
7. Refine and polish

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?

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

- Good names make code understandable with minimal context. You should strive for self-explanatory names

More examples

- We want to write a function, called both_na(), that counts how many positions in two input 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 inputs x and y.

Tip: Search for existing functionality to get us started…

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() function
is.na(x)
[1] FALSE FALSE  TRUE FALSE  TRUE
```

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() function
is.na(x)
[1] FALSE FALSE  TRUE FALSE  TRUE
```


Q. How many TRUE values are there?

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() function
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")
  }

  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) )
}
```

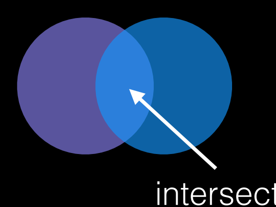
Re-cap: Steps for function writing

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

Break!

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")

```r
# Start with a simple version of the 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)
```


Follow along!

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

# Start with a simple version of the 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)

df1

	IDs	exp
1	gene1	2
2	gene2	1
3	gene3	1

df2

	IDs	exp
1	gene2	-2
2	gene4	NA
3	gene3	1
4	gene5	2

Follow along!

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

# Start with a simple version of the 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)


```r
# Simplify further to single vectors
x <- df1$IDs
y <- df2$IDs

# Now what do we do?
```

x

	IDs	exp
1	gene1	2
2	gene2	1
3	gene3	1

y

	IDs	exp
1	gene2	-2
2	gene4	NA
3	gene3	1
4	gene5	2

Follow along!

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

# Start with a simple version of the 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?

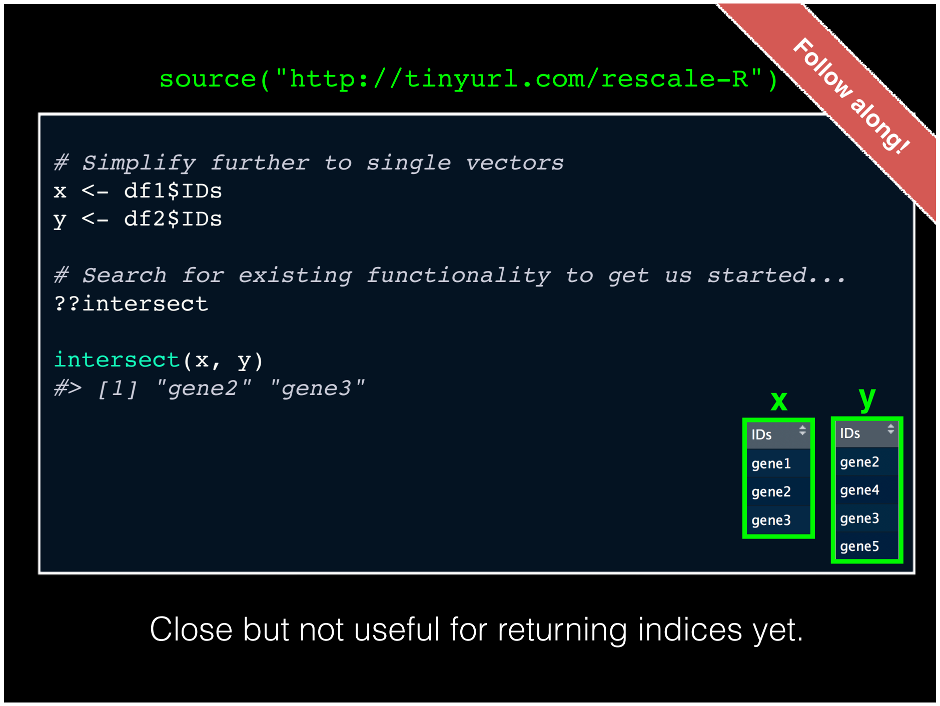
x

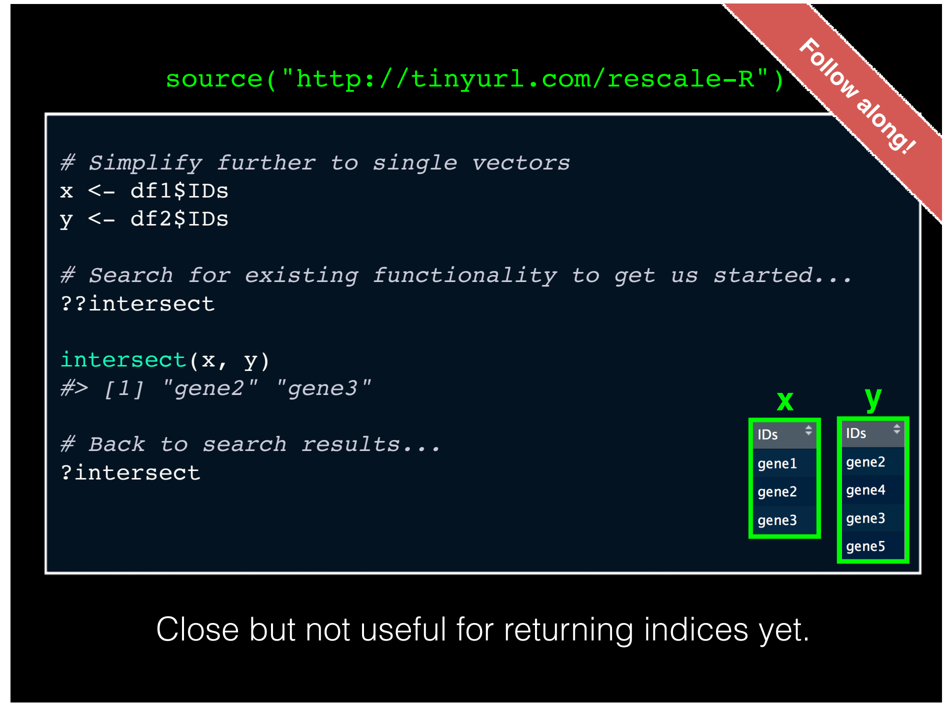
IDs
gene1
gene2
gene3

y

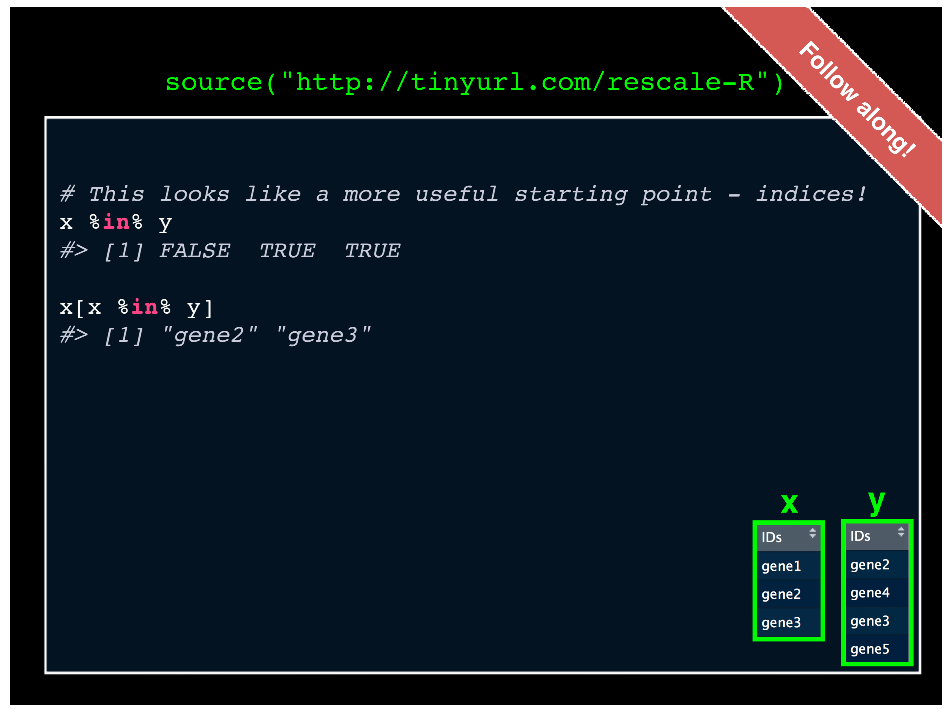
IDs
gene2
gene4
gene3
gene5

Tip: Search for existing functionality to get us started…


source("http://tinyurl.com/rescale-R")
Follow along!
# Simplify further to single vectors
x <- df1$IDs
y <- df2$IDs
# Search for existing functionality to get us started...
??intersect
intersect(x, y)
#> [1] "gene2" "gene3"
x
IDs
gene1
gene2
gene3
y
IDs
gene2
gene4
gene3
gene5
Close but not useful for returning indices yet.


source("http://tinyurl.com/rescale-R")
Follow along!
# Simplify further to single vectors
x <- df1$IDs
y <- df2$IDs
# Search for existing functionality to get us started...
??intersect
intersect(x, y)
#> [1] "gene2" "gene3"
Back to search results...
?intersect
x
IDs
gene1
gene2
gene3
y
IDs
gene2
gene4
gene3
gene5
Close but not useful for returning indices yet.


source("http://tinyurl.com/rescale-R")
Follow along!
This looks like a more useful starting point - indices!
x %in% y
#> [1] FALSE TRUE TRUE
x
IDs
gene1
gene2
gene3
y
IDs
gene2
gene4
gene3
gene5


source("http://tinyurl.com/rescale-R")
Follow along!
This looks like a more useful starting point - indices!
x %in% y
#> [1] FALSE TRUE TRUE
x[x %in% y]
#> [1] "gene2" "gene3"
x
IDs
gene1
gene2
gene3
y
IDs
gene2
gene4
gene3
gene5

Follow along!

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

```r
# 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...
cbind( x[ x %in% y ], y[ y %in% x ] )
#>      [,1]    [,2]
#> [1,] "gene2" "gene2"
#> [2,] "gene3" "gene3"
```

x

IDs
gene1
gene2
gene3

y

IDs
gene2
gene4
gene3
gene5

Follow along!

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

```r
# 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"
```

Working Snippet!

```r
# We can now cbind() these these results...
cbind( x[ x %in% y ], y[ y %in% x ] )
#>      [,1]    [,2]
#> [1,] "gene2" "gene2"
#> [2,] "gene3" "gene3"
```

x

IDs
gene1
gene2
gene3

y

IDs
gene2
gene4
gene3
gene5

Follow along!

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

```r
# Our previous working snippet...
cbind( x[ x %in% y ], y[ y %in% x ] )

# Make this snippet 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"
```

x

IDs
gene1
gene2
gene3

y

IDs
gene2
gene4
gene3
gene5

Follow along!

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

```r
# Our previous working snippet...
cbind( x[ x %in% y ], y[ y %in% x ] )

# Make this snippet 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"
```

df1

	IDs	exp
1	gene1	2
2	gene2	1
3	gene3	1

df2

	IDs	exp
1	gene2	-2
2	gene4	NA
3	gene3	1
4	gene5	2

Follow along!

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

```r
# Lets edit 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
```

df1

	IDs	exp
1	gene1	2
2	gene2	1
3	gene3	1

df2

	IDs	exp
1	gene2	-2
2	gene4	NA
3	gene3	1
4	gene5	2

Follow along!

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

```r
# Lets edit 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
```

N.B. Our input $IDs column name may change:

So lets add flexibility by allowing the user to specify the gene containing column name

Follow along!

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

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

Follow along!

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

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

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
```

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

Follow along!

```r
# Improve by simplifying for for human consumption
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"] )
}
```

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

Follow along!

```r
# Improve by simplifying for for human consumption
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
```

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

Follow along!

```r
# 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 in data.frame(..., check.names = FALSE): row
names were found from a short variable and have been
discarded
```

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

Follow along!

```r
# 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
```

For more details refer to sections 2-5 in last days handout!

https://bioboot.github.io/bggn213_S19/lectures/#6

Remember **Section 1B** (question 6) is your last days homework (see also scoring rubric).

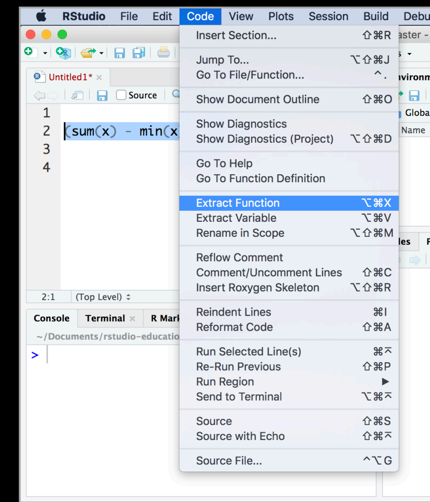
The **Sections 2** to **5** are there for your benefit.

RStudio Tip: Extract function

- You can use the Extract Function feature of the RStudio IDE to turn a piece of code into a function.
- **Code > Extract Function**
- Calculate the average grade dropping the lowest score from these two students:

```
c(100, 100, 100, 100,
  100, 100, 100, 90)
```

```
c(100, 90, 90, 90,
   90, 90, 97, 80)
```



CRAN & **Bioconductor**

R Highlight!

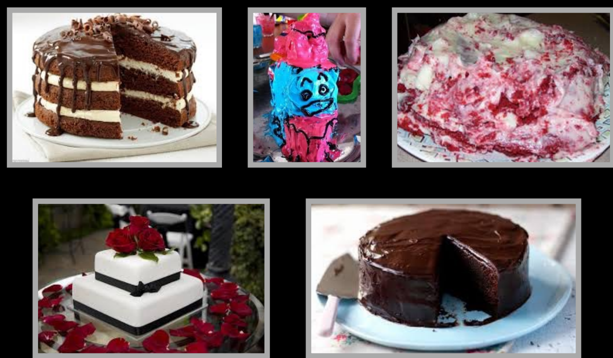
Major repositories for **R packages** that extend R functionality

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 ~14,038 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



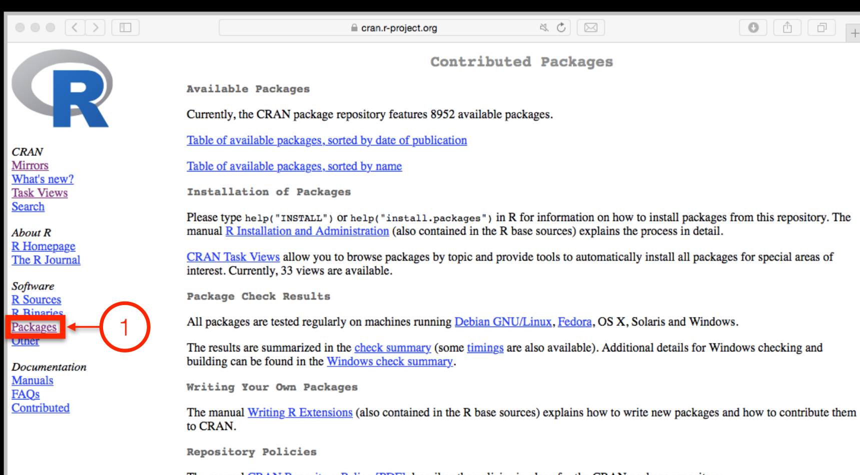
R packages can be of variable quality and often there are multiple packages with overlapping functionality.

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



cran.r-project.org

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

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.

Installing a package

RStudio > Tools > Install Packages

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

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 3.8 had 1,649 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

"Accessible, curated metagenomic data through ExperimentHub"
Pasolli *et al.*
Nature Methods 2017, 14:1023–1024.

Installing a bioconductor package

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

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

Installing a bioconductor package

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

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

Do it Yourself!

Your Turn: Form a group of 3, pick a package to explore and install, Report back to the class.

ggplot2, **bio3d**, rmarkdown, rgl, **dplyr**, rentrez, **reprex** **blogdown**, **shiny**, **msa**, flexdashborad, phyloseq,

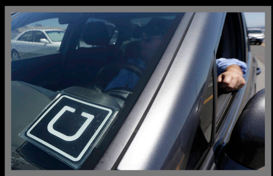
Questions to answer:

- 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?

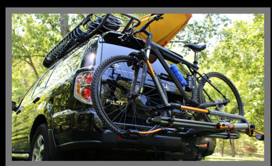
[Collaborative Google Doc Link] <- See Website

Key Idea:

Using **existing base functions** in R is like riding a bus - it is relatively straightforward you just need to know which bus to use and know where to get on and get off.



Being able to use **CRAN & bioconductor** packages and functions is like having access to UBER - they can take you more places but may only cover big cities.



Writing your **own functions** in R is like driving an SUV with kayak & bike on top - it takes more work, you need to know how to get there. Ultimately, however it will give you the flexibility to go completely new places!

Learning Resources

- **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/ >

[Muddy Point Assessment Link]