

Introduction to Computational Statistics with R

Stats 102A

Miles Chen

Acknowledgements: Michael Tsiang and Hadley Wickham

Week 1 Wednesday



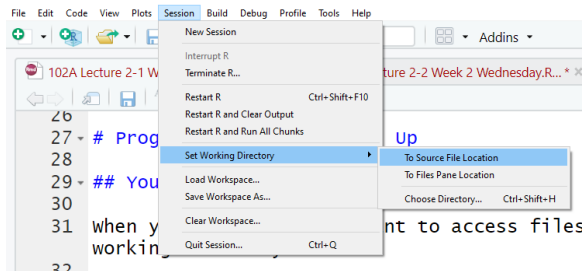
Section 1

Programming: Set Up and Shortcuts

Your Working Directory

When you use R, you may want to read files or output to files. Commands given to R will search the working directory.

When I work a file, I will often use the menu: **Session > Set Working Directory > To Source File Location**.



Your Working Directory

You can also check the working directory with `getwd()`. You can set it with `setwd("path/to/directory/")`

Note that on Windows machines, you must use a forward slash `/` or escaped backslashes `\\` to indicate the path to a directory.

`"C:\Users\Miles\Documents"` will not work.

`"C:/Users/Miles/Documents"` and `"C:\\Users\\Miles\\Documents"` will work.

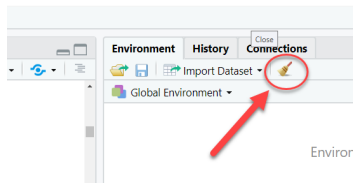
Note: If you are working on a file with another person, you should NEVER set the working directory to something specific on your own computer.

Checking and Clearing the environment

As you work, your environment, or workspace, will fill with objects. You can check the environment for a list of its contents with `ls()`. You can remove individual elements with `rm()`. `rm(x)` will remove the variable `x`.

You should get in the habit of clearing out your environment whenever you start working on a new file or project. Keep in mind that whenever you render/knit a markdown file, R begins with an empty environment.

You can clear the environment this by clicking the Broom Icon in the environment panel, or by typing:



Checking and Clearing the environment

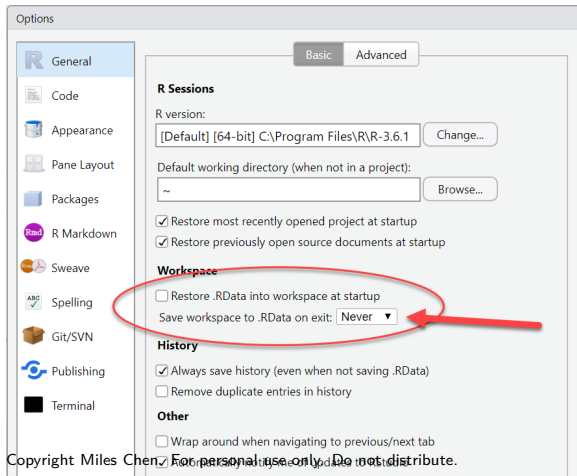
Another quick command to clear the workspace is to type:

```
rm(list = ls())
```

```
# ls() provides a vector of all the object names in the environment.  
# rm() goes through and removes all of those names
```

Recommendation: Don't save your workspace when you exit RStudio

I **highly** recommend changing the following option: Tools > Global Options ... On the General tab, uncheck “Restore .RData into workspace at startup” and change the setting ‘Save Workspace to .Rdata on Exit’ to ‘Never’



Recommendation: Don't save your workspace when you exit RStudio

Not saving on exit forces you to have good save habits.

It also prevents issues where your script receives old and incorrect values when it refers to common names such as `x` or `results`. If you don't save your workspace on exit, those old values will simply not exist when you start RStudio the next time.

If you ever want to save your workspace, you can do so explicitly and you will have the workspace available as a labeled file that you can transport.

Toggle Comments in your code

You already know you can make a comment in your code with #

Commenting can be useful method to quickly disable lines of code.

In R Studio, the keyboard shortcut **Ctrl/Cmd + Shift + C** will toggle a line to be a comment. You can toggle entire blocks of code this way as well.

Writing Scripts in RStudio

Quickly create a new R Script file, use the keyboard shortcut **Ctrl/Cmd + Shift + N**

This will create a blank plain text document (with extension .R)

Each line or a block of highlighted text can then be executed with **Ctrl/Cmd + Enter**.

If you want to run the entire script, you can source the document: `source("script.R")`

Keyboard shortcut to source the entire script: **Ctrl/Cmd + Shift + S**

Other useful shortcuts:

Clear the console: **Ctrl/Cmd + L**

Restart your R Session: **Ctrl/Cmd + Shift + F10**

Knit your current Rmd File: **Ctrl/Cmd + Shift + K**

Section 2

Basic Data Structures

The most important family of objects in R is **vectors**.

There are two types of vectors: atomic vectors and generic vectors (also called lists).

Question: What is the main difference between an atomic vector and a list?

Atomic Vectors

The most fundamental object in R is an **atomic vector** (or vector), which is an ordered collection of values.

Atomic vectors have six basic types (though we will only work with the first four):

<code>typeof()</code>	<code>mode()</code>
logical	logical
double	numeric
integer	numeric
character	character
complex	complex
raw	raw

All elements of an atomic vector must be of the same type.

Doubles and Integers

The conversion between integer and double values is often done automatically in R, so the distinction is typically not needed (they are both numeric types).

There is an important difference: doubles are numeric values stored with floating point precision, while integers are stored as exact integer values.

The default numeric type is double. Integers can be indicated with an L after the number and vectors of integers can be created using the colon (:) operator.

Doubles and Integers

```
typeof(c(1, 2, 3))
```

```
## [1] "double"
```

```
typeof(1:3)
```

```
## [1] "integer"
```

```
is.double(1)
```

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

```
is.integer(1L)
```

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


Lists

A **list** (or generic vector) is an ordered collection of objects. Lists are the most flexible objects in R, as each component in a list can be *any* other object, including other lists.

The most common objects built from vectors are summarized in the table below.

Dimension	Homogeneous	Heterogeneous
1-dim	Atomic vector	List (generic vector)
2-dim	Matrix	Data frame
n -dim	Array	

Every vector (atomic or generic) can also have **attributes**, which is a named *list* of arbitrary metadata.

Two attributes are particularly important: The **dimension** attribute turns vectors into matrices and arrays, and the **class** attribute develops the **S3** object system (which we will cover later in the course).

Self-quiz (when you study later)

- How do you get and set attributes of an object?
- How is a matrix different from a data frame? How is each one internally stored in R?
- What is a factor? How is a factor internally stored in R?

Attributes

The `attr()` function can be used to get or set single attributes of an object.

The `attributes()` function can be used to access the entire list of attributes.

Let's take a look at the data frame `trees`.

```
head(trees)
```

##	Girth	Height	Volume
## 1	8.3	70	10.3
## 2	8.6	65	10.3
## 3	8.8	63	10.2
## 4	10.5	72	16.4
## 5	10.7	81	18.8
## 6	10.8	83	19.7

Attributes

The attributes of the trees data frame is a list with three elements: names (the column names), class (data.frame), and row names

```
attributes(trees)
```

```
## $names
## [1] "Girth"  "Height" "Volume"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
## [23] 23 24 25 26 27 28 29 30 31
```

Attributes

We can technically add any arbitrary bit of information and throw it into the attributes list and then retrieve that information later. This is not a normal practice, but because attributes are simply a list, it's possible.

```
attr(trees, "info") <- "This data frame is about trees!!"  
attributes(trees)
```

```
## $names  
## [1] "Girth" "Height" "Volume"  
##  
## $class  
## [1] "data.frame"  
##  
## $row.names  
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22  
## [23] 23 24 25 26 27 28 29 30 31  
##  
## $info  
## [1] "This data frame is about trees!!"
```

Matrices

A **matrix** in R is an atomic vector with a dimension attribute of length 2.

```
M <- 1:10  
M # M is an atomic vector of integers
```

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

```
class(M)
```

```
## [1] "integer"
```

```
attr(M, "dim") <- c(2, 5) # I set dimension attributes  
M # M is now a matrix of integers
```

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

Matrices

```
attributes(M) # there's only one attribute: dim
```

```
## $dim  
## [1] 2 5
```

```
class(M) # class is smart enough to figure out that it's a matrix
```

```
## [1] "matrix" "array"
```

```
attr(M, "dim") <- NULL # remove the dimension attribute  
M # M is back to a vector
```

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

```
class(M)
```

```
## [1] "integer"
```

Arrays

An **array** in R is a vector with a dimension attribute of length more than 2.

```
A <- 1:12
attr(A, "dim") <- c(2, 3, 2)
A
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Arrays can also be created using `array()`.

Data Frame

A **data frame** in R is internally stored as a list of equal length vectors with a class attribute called `data.frame`.

```
head(trees, 4)
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
## 4  10.5     72   16.4
```

```
class(trees)
```

```
## [1] "data.frame"
```

```
typeof(trees)
```

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

Factors

A **factor** is a vector used to represent categorical values. It is internally stored as an integer vector with levels and class attributes.

```
gender <- c("M", "F", "F", "X", "M", "F")  
gender_fac <- factor(gender)  
gender_fac
```

```
## [1] M F F X M F  
## Levels: F M X
```

```
levels(gender_fac)
```

```
## [1] "F" "M" "X"
```

```
typeof(gender_fac)
```

```
## [1] "integer"
```

Factors

Internally, the factor is an integer vector. When displayed, it replaces the integer with the corresponding level.

```
gender_fac
```

```
## [1] M F F X M F  
## Levels: F M X
```

```
as.integer(gender_fac)
```

```
## [1] 2 1 1 3 2 1
```

```
attributes(gender_fac)
```

```
## $levels  
## [1] "F" "M" "X"  
##  
## $class  
## [1] "factor"
```

Factors

Watch out! If a vector of numbers get turned into factors, the unique values get stored as levels. This can lead to unexpected results if you aren't careful.

```
x <- c(0, 1, 10, 5)
x_fac <- factor(x)
x_fac
```

```
## [1] 0  1 10 5
## Levels: 0 1 5 10
```

```
mean(x_fac) # we try to take the mean but it doesn't work
```

```
## Warning in mean.default(x_fac): argument is not numeric or logical:
## returning NA
```

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

Factors

```
# so we coerce to numeric, but the result doesn't make sense  
mean(as.numeric(x_fac)) # the mean of 0, 1, 10, 5 should be 4
```

```
## [1] 2.5
```

```
as.numeric(x_fac) # internally, they are stored as integers
```

```
## [1] 1 2 4 3
```

```
x_fac # again, x_fac is a factor
```

```
## [1] 0 1 10 5  
## Levels: 0 1 5 10
```

```
mean(as.numeric(as.character(x_fac))) # this works
```

```
## [1] 4
```

Factors - other rules

You can't meaningfully combine factors with `c()`

```
c(factor(c("a","b")), factor(c("b","c")))
```

```
## [1] 1 2 1 2
```

You can't use values that are not in the levels

```
gender_fac[2] <- "male"
```

```
## Warning in '[<-.factor'('*tmp*', 2, value = "male"): invalid factor  
## level, NA generated
```

```
gender_fac
```

```
## [1] M    <NA> F    X    M    F
```

```
## Levels: F M X
```

Coercion

To illustrate the idea of coercion, we create the following vectors.

```
l <- c(TRUE, FALSE)
i <- 1L
d <- c(5, 6, 7)
ch <- c("a", "b")
```

Atomic vectors in R can only contain one data type. When values of different types are combined into a single vector, the values are **coerced** into a single type.

Question: What is the output for the following commands?

```
typeof(c(l, i, d))
typeof(c(l, d, ch))
```

Coercion

Coercion looks at the least restrictive type and coerces everything to that type.

The order from most restrictive to least restrictive is logical < integer < double < character < list

```
c(1, i, d)
```

```
## [1] 1 0 1 5 6 7
```

```
typeof(c(1, i, d))
```

```
## [1] "double"
```

```
c(1, i, ch)
```

```
## [1] "TRUE" "FALSE" "1" "a" "b"
```

```
typeof(c(1, d, ch))
```

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


Implicit Coercion

Coercion often happens automatically. Most mathematical functions (+, log(), abs(), etc.) will coerce to a double or integer, and most logical operations (&, |, any(), etc.) will coerce to a logical.

```
trials <- c(FALSE, FALSE, TRUE)
as.numeric(trials)
```

```
## [1] 0 0 1
```

```
sum(trials)  # Total number of TRUEs
```

```
## [1] 1
```

```
mean(trials) # Proportion that are TRUE
```

```
## [1] 0.3333333
```

Explicit Coercion

The `as` functions can be used to explicitly coerce objects, if possible.

```
as.character(trials)
```

```
## [1] "FALSE" "FALSE" "TRUE"
```

```
as.logical(c(0, 1))
```

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

```
as.numeric("cat")
```

```
## Warning: NAs introduced by coercion
```

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

Explicit Coercion

Explicit coercion rules

```
# anything numeric that is not 0 becomes TRUE except NaN becomes NA  
as.logical(c(0, 1, -1, 0.1, 2, -Inf, 2.2e-308, NaN) )
```

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

```
# accepted spellings of logical values  
as.logical(c("F", "FALSE", "False", "false", "T", "TRUE", "True", "true"))
```

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

```
as.logical(c("f","t", "cat", 0, 1)) # other characters not coerced
```

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

Question: What is the difference between NA, NULL, and NaN?

Question: What is the difference between NA, NULL, and NaN?

- NA is used to represent missing or unknown values. There are NA for each type.
- NULL is used to represent a empty or nonexistent value. NULL is its own type.
- NaN is type double and is used to represent indeterminate forms in mathematics (such as $0/0$ or $-\text{Inf} + \text{Inf}$).

Including NA in an atomic vector or matrix will not change the data type. Internally, R has an NA for each data type.

NA

NA_integer_

NA_real_

NA_character_

To check for NA, you must use the function `is.na()`. You cannot use `==`

```
NA == NA
```

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

```
is.na(NA)
```

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

NULL

R uses NULL to represent the NULL object. It is its own type.

```
typeof(NULL)
```

```
## [1] "NULL"
```

```
is.null(NULL)
```

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

```
is.na(NULL)
```

```
## logical(0)
```


NULL

```
is.logical(NULL)
```

```
## [1] FALSE
```

```
NULL + FALSE # operations with NULL result in a length 0 vector
```

```
## integer(0)
```

```
c(4, 5, NULL, 3) # "including" NULL is like including nothing
```

```
## [1] 4 5 3
```

```
NULL == NULL
```

```
## logical(0)
```

Vector Arithmetic

Arithmetic can be done on numeric vectors using the usual arithmetic operations. The operations are **vectorized**, i.e., they are applied elementwise (to each individual element).

```
x <- c(1, 2, 3)
y <- c(100, 200, 300)
x + y
```

```
## [1] 101 202 303
```

```
x * y
```

```
## [1] 100 400 900
```

Vector Recycling

When applying arithmetic operations to two vectors of different lengths, R will automatically **recycle**, or repeat, the shorter vector until it is long enough to match the longer vector.

Question: What is the output of the following commands?

```
c(1, 2, 3) + c(100, 200, 300, 400, 500, 600)  
c(1, 2, 3) + c(100, 200, 300, 400, 500)
```

Question: When will R throw a warning when recycling?

Vector Recycling

```
c(1, 2, 3) + c(100, 200, 300, 400, 500, 600)
```

```
## [1] 101 202 303 401 502 603
```

```
c(1, 2, 3) + c(100, 200, 300, 400, 500)
```

```
## Warning in c(1, 2, 3) + c(100, 200, 300, 400, 500): longer object  
## length is not a multiple of shorter object length
```

```
## [1] 101 202 303 401 502
```

Vector Recycling - Matrices

```
M <- rbind(c( 1,  2, 3),  
           c( 4,  5, 6),  
           c( 7,  8, 9),  
           c(10, 11, 12))  
print(M)
```

```
##      [,1] [,2] [,3]  
## [1,]    1    2    3  
## [2,]    4    5    6  
## [3,]    7    8    9  
## [4,]   10   11   12
```

```
x <- c(100, 200, 300)  
M + x # recycling is done column-wise
```

```
##      [,1] [,2] [,3]  
## [1,]  101  202  303  
## [2,]  204  305  106  
## [3,]  307  108  209  
## [4,]  110  211  312
```

Vector Recycling - Matrices

```
t(M) # t() transposes the matrix
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]    1    4    7   10  
## [2,]    2    5    8   11  
## [3,]    3    6    9   12
```

```
t(M) + x # recycling is still done column-wise
```

```
##      [,1] [,2] [,3] [,4]  
## [1,]  101  104  107  110  
## [2,]  202  205  208  211  
## [3,]  303  306  309  312
```

```
t(t(M) + x) # transposing the result is effectively equivalent to recycling row-wise
```

```
##      [,1] [,2] [,3]  
## [1,]  101  202  303  
## [2,]  104  205  306  
## [3,]  107  208  309  
## [4,]  110  211  312
```