Introduction to Computational Statistics with R

Week 1 Wednesday

Miles Chen, PhD

Department of Statistics

Section 1

Programming: Set Up and Shortcuts

Tip: create a separate folder for each hw assignment

- Tip: Create a separate folder for each homework assignment.
- Tip: do not use spaces in folder names, use underscores or dashes instead
- Put all related files into the same folder

For HW1, create a folder on your computer called "HW1".

Download the three files for the assignment (`102A_hw_01_instructions.pdf`, `102A_hw_01_output.qmd`, and `month_names.txt`) and put them into the same HW1 folder.

Do not work directly from your downloads folder.

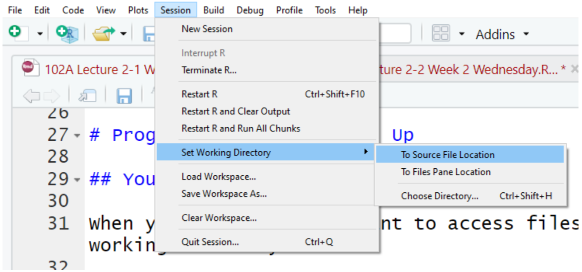
Navigate to your HW1 folder and open the Qmd file from there.

You will need to create a new file in the folder: `102a_hw_01_script_Joe_Bruin.R`

Your Working Directory

To make sure R can see the appropriate files, you need to set the working directory.

I use the menu commands to set my working directory: **Session** > **Set Working Directory** > **To Source File Location**.



```
# Prog                Up

## You

When y                nt to access files
workin
```

Your Working Directory

At some point, you might encounter the following message or something like it:

```
Warning: cannot open file 'month_names.txt': No such file or directory
Error in file(file, "rt") : cannot open the connection
```

This means R is searching for a file "month_names.txt" in the working directory and it does not see it.

Either:

- the file is saved somewhere else on your computer (not in the same directory as the homework Qmd file)
- your working directory is set to the wrong folder location

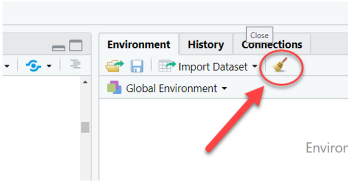
Fix the problem and try again.

Checking and Clearing the environment

As you work, your environment, or workspace, will fill with objects. You can check the environment pane in R Studio to see what exists. You may also get a list of the 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 a markdown file, R begins with an empty environment.

You can clear the environment this by clicking the Broom Icon in the environment pane.



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
```

Problem: “My code runs in R, but when I try to render, I get an error.”

“My code runs in R, but when I try to Render, I get an error.”

This is one of the most common errors students encounter. There are many possible reason this happens, but one of the most common scenarios is this:

1. Student creates an object in R.
2. Student modifies or deletes the line of code that creates the original object.
3. The object still exists in the Environment, so new lines of code work
4. When it comes time to render, the line to create said object doesn’t exist or creates a different version of the object. R is not able to execute the other lines of code.

Resolving the Issue

1. Save your files: qmd file, R script, etc.
2. Clear your environment.
3. Starting at the beginning of your qmd file, run each code chunk one at a time.
   - Most likely, you will encounter an error message.
   - See which line caused the error. This will provide some insight into which object is missing or needs to be fixed.
4. If you can clear the environment and run through every single code chunk with no errors, then your file should render without problems.

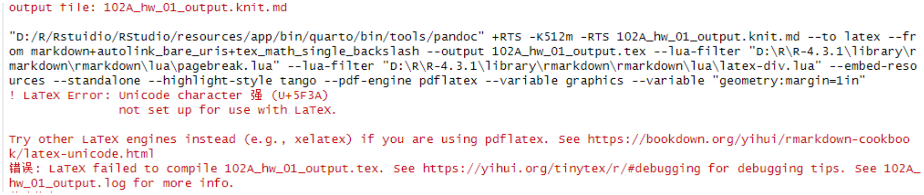
Resolving Really Stubborn Issues

If the steps in the previous slide do not work, you can attempt the following to try to resolve really stubborn issues. This is much more work, so I only recommend it when absolutely necessary.

1. Save your files: the current Qmd file that you can't render, R script, etc. I'll refer to the Qmd file with errors as ProblemQmd.
2. Clear your environment.
3. Create a new qmd file and save it to the same Homework folder. I will call the FreshQmd.
4. render the FreshQmd file. Because it is brand new, it should render with no problems. Delete any unnecessary stuff from FreshQmd.
5. Copy one paragraph or one code chunk from ProblemQmd to FreshQmd. After copying, render FreshQmd. It should render.
6. Continue copying one paragraph or one code chunk at a time from ProblemQmd to FreshQmd and render FreshQmd each time.
7. When you copy something from ProblemQmd to FreshQmd and FreshQmd encounters and error when rendering, you know the problem exists in the most recently copied chunk. You can event try deleting that chunk and rendering to verify that it is indeed that piece.
8. Resolve the issue.

Tip: Set your language to English

If your language is not English, error messages will contain foreign characters. LaTeX is not able to handle these characters and will result in an error like the following.



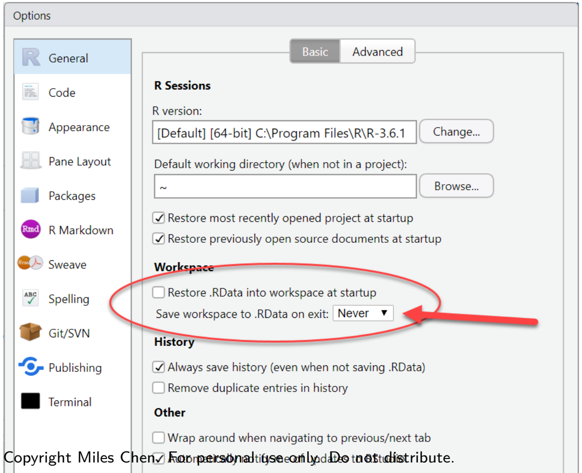
```
output file: 102A_hw_01_output.knit.md

"D:/R/Rstuidio/RStudio/resources/app/bin/quarto/bin/tools/pandoc" +RTS -K512m -RTS 102A_hw_01_output.knit.md --to latex --fr
om markdown+autolink_bare_uris+tex_math_single_backslash --output 102A_hw_01_output.tex --lua-filter "D:\R\R-4.3.1\library\r
markdown\rmarkdown\lua\pagebreak.lua" --lua-filter "D:\R\R-4.3.1\library\rmarkdown\rmarkdown\lua\latex-div.lua" --embed-reso
urces --standalone --highlight-style tango --pdf-engine pdflatex --variable graphics --variable "geometry:margin=1in"
! LaTeX Error: Unicode character 强 (U+5F3A)
               not set up for use with LaTeX.

Try other LaTeX engines instead (e.g., xelatex) if you are using pdflatex. See https://bookdown.org/yihui/rmarkdown-cookboo
k/latex-unicode.html
错误: LaTeX failed to compile 102A_hw_01_output.tex. See https://yihui.org/tinytex/r/#debugging for debugging tips. See 102A_
hw_01_output.log for more info.
```

How to set your language to English:

- For Windows: https://stackoverflow.com/a/26674852
- For Mac: https://stackoverflow.com/a/28353147

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

Tip: Change the following option: Tools > Global Options ... On the General tab, uncheck "Restore .RData into workspace at startup" and change the setting 'Save Worskspace to .Rdata on Exit' to 'Never'



R Sessions

R version: [Default] [64-bit] C:\Program Files\R\R-3.6.1

Default working directory (when not in a project): ~

- [✓] Restore most recently opened project at startup
- [✓] Restore previously open source documents at startup

Workspace

- [ ] Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

History

- [✓] Always save history (even when not saving .RData)
- [ ] Remove duplicate entries in history

Other

- [ ] Wrap around when navigating to previous/next tab
- [✓] Automatically notify me of updates to RStudio

Tip: 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**

render your current Qmd File: **Ctrl/Cmd + Shift + K**

Section 2

Basic Data Structures

Related Reading

Advanced R, 1st Edition, Chapter: Data Structures

http://adv-r.had.co.nz/Data-structures.html

Extended, optional reading:

Advanced R, 2nd edition, Chapter: Vectors

https://adv-r.hadley.nz/vectors-chap.html

Vectors

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):

`typeof()`	`mode()`
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

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

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

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

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

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

```
[1] TRUE
```

```r
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	

Attributes

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] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[31] 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.

```r
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] 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[31] 31

$info
[1] "This data frame is about trees!!"
```

Matrices

A **matrix** in R is an atomic vector with a dimension attribute: a vector for the number of rows and columns.

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

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

```r
class(M)
```

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

```r
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

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

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

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

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

```r
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
```

```r
class(M)
```

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

Arrays

An **array** in R is an atomice vector where the dimension attribute is a vector longer 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.

```r
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
```

```r
levels(gender_fac)
```

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

```r
typeof(gender_fac)
```

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

Factors

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

```r
gender_fac
```

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

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

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

```r
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.

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

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

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

```
[1] NA
```

Factors

```r
# 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
```

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

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

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

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

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

```
[1] 4
```

Factors - other rules

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

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

```r
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?

```r
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

```r
c(l, i, d)
```

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

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

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

```r
c(l, i, ch)
```

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

```r
typeof(c(l, 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.

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

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

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

```
[1] 1
```

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

```
[1] 0.3333333
```

Explicit Coercion

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

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

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

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

```
[1] FALSE  TRUE
```

```r
as.numeric("dog")
```

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

```
[1] NA
```

Explicit Coercion

Explicit coercion rules

```r
# 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
[8]    NA
```

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

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

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

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

Special Values

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

Special Values

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 an 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 `-Inf + Inf`).

Example: You are storing information about people. You have columns for their name, their age, their partner's name, and their partner's age.

Joe is a person. You don't know Joe's age. You enter `NA` for Joe's age. (The age exists, but you don't know it.) Joe does not have a partner. You would enter `NULL` for the partner's name and partner's age. (These values do not exist.) If Joe had a partner but you did not know the partner's age, you would enter `NA` instead of `NULL`.

NA

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

```
NA
NA_integer_
NA_real_
NA_character_
```

NA

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

```r
NA == NA
```

```
[1] NA
```

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

```
[1] TRUE
```

NULL

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

```r
typeof(NULL)
```

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

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

```
[1] TRUE
```

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

```
logical(0)
```

NULL

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

```
[1] FALSE
```

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

```
integer(0)
```

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

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

```r
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

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

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

```r
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

```r
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
```

```r
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
```