Introduction to R

"How do you turn this thing on?"

Samuel Robinson, Ph.D.

Sep. 4 2023

Motivation

- “Why do I need to learn R?”

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*
  - Writing presentations and papers

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*
  - Writing presentations and papers
  - *Keeping a record of what you've done*

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*
  - Writing presentations and papers
  - *Keeping a record of what you've done*
- "What is R bad at?"

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*
  - Writing presentations and papers
  - *Keeping a record of what you've done*
- "What is R bad at?"
  - No point-and-click interface; simple things can take more time

Motivation

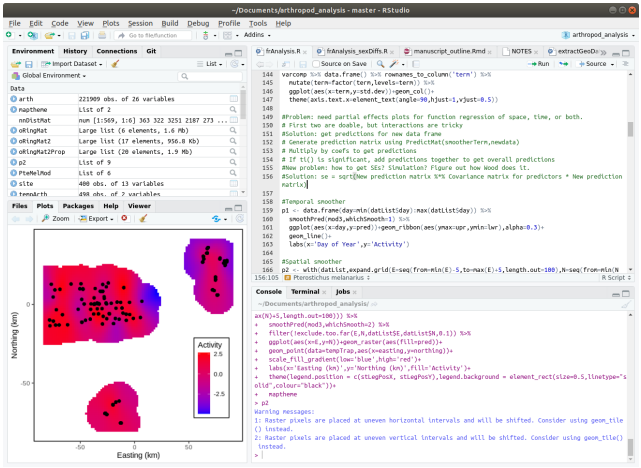
- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*
  - Writing presentations and papers
  - *Keeping a record of what you've done*
- "What is R bad at?"
  - No point-and-click interface; simple things can take more time
  - Can be slow if datasets are large*

Motivation

- "Why do I need to learn R?"
  - Free, powerful, and very common
  - Interfaces with other languages (e.g. C++), and can help you learn other languages
- "What is R good at?"
  - Displaying data, running models, and processing data*
  - Writing presentations and papers
  - *Keeping a record of what you've done*
- "What is R bad at?"
  - No point-and-click interface; simple things can take more time
  - Can be slow if datasets are large*
- I am not here to teach you programming, but some basic techniques are useful

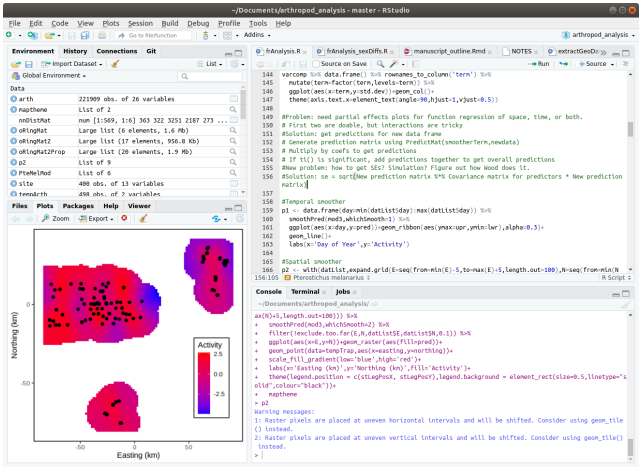
RStudio GUI

- The **Console** is the main input into R (where you tell it to do things)



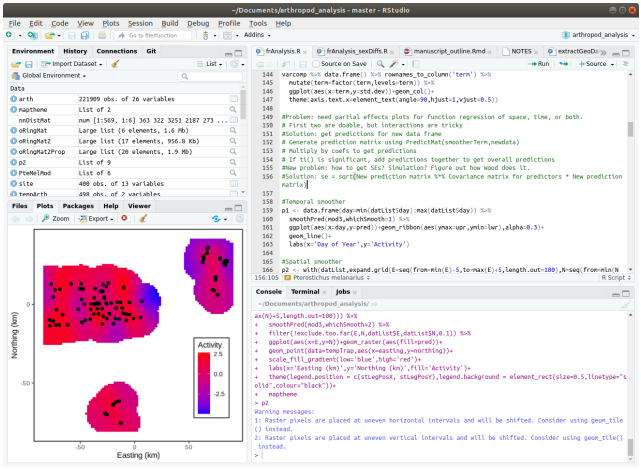
RStudio GUI

- The **Console** is the main input into R (where you tell it to do things)
- **Scripts** are lists of commands that get passed into the console



RStudio GUI

- The **Console** is the main input into R (where you tell it to do things)
- **Scripts** are lists of commands that get passed into the console
- If you're using RStudio, 2 of the 4 panes will be dedicated to the console and scripts



Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices
  - Dataframes & Lists

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices
  - Dataframes & Lists
- Some common **functions** (things done to objects):

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices
  - Dataframes & Lists
- Some common **functions** (things done to objects):
  - *mean*, *sd*, *median*, *quantile*, *c*, *paste*

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices
  - Dataframes & Lists
- Some common **functions** (things done to objects):
  - *mean*, *sd*, *median*, *quantile*, *c*, *paste*
  - *plot*, *summary*

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices
  - Dataframes & Lists
- Some common **functions** (things done to objects):
  - *mean*, *sd*, *median*, *quantile*, *c*, *paste*
  - *plot*, *summary*
    - these are polymorphic functions: they do different things to different types of objects

Objects and Functions

- Everything in R is either an **Object** or a **Function**. All must have a unique name, or else the *Steve Problem** occurs.
- Some common **objects** (things stored in memory):
  - Characters, Logicals, & Numerics
  - Vectors & Matrices
  - Dataframes & Lists
- Some common **functions** (things done to objects):
  - *mean*, *sd*, *median*, *quantile*, *c*, *paste*
  - *plot*, *summary*
    - these are polymorphic functions: they do different things to different types of objects
  - Control flow - *if* and *for*

Objects

- Let's make some objects. These are all single objects:

```
myString <- "Hello world" #A string object
myNumeric <- 12345 #A numeric object
myLogical <- TRUE #A logical object
```

Objects

- Let's make some objects. These are all single objects:

```
myString <- "Hello world" #A string object
myNumeric <- 12345 #A numeric object
myLogical <- TRUE #A logical object
```

- These are objects joined into a *vector*, joined by the function c (concatenate):

```
myCharVec <- c("I like pie","I like cake","I like anything you bake")
myNumVec <- c(1,2,3,4,5)
myLogVec <- c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE)
```

Objects

- Let's make some objects. These are all single objects:

```
myString <- "Hello world" #A string object
myNumeric <- 12345 #A numeric object
myLogical <- TRUE #A logical object
```

- These are objects joined into a *vector*, joined by the function c (concatenate):

```
myCharVec <- c("I like pie","I like cake","I like anything you bake")
myNumVec <- c(1,2,3,4,5)
myLogVec <- c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE)
```

- How long are each of these vectors?

```
howLong <- c(length(myCharVec), length(myNumVec), length(myLogVec))
howLong #This executes the `print` command on `howLong`
```

```
## [1] 3 5 6
```

Vectors - “getting”

```
myCharVec #Here's what's inside the whole thing
```

```
## [1] "I like pie"                "I like cake"
## [3] "I like anything you bake"
```

Vectors - “getting”

```
myCharVec #Here's what's inside the whole thing
```

```
## [1] "I like pie"                "I like cake"
## [3] "I like anything you bake"
```

- Single number:

```
myCharVec[1]
```

```
## [1] "I like pie"
```

Vectors - "getting"

```
myCharVec #Here's what's inside the whole thing
```

```
## [1] "I like pie"                "I like cake"
## [3] "I like anything you bake"
```

- Single number:

```
myCharVec[1]
```

```
## [1] "I like pie"
```

- Vector of numbers

```
myCharVec[c(2,3)]
```

```
## [1] "I like cake"               "I like anything you bake"
```

Vectors - "getting"

```
myCharVec #Here's what's inside the whole thing
```

```
## [1] "I like pie"                "I like cake"
## [3] "I like anything you bake"
```

- Single number:

```
myCharVec[1]
```

```
## [1] "I like pie"
```

- Vector of numbers

```
myCharVec[c(2,3)]
```

```
## [1] "I like cake"               "I like anything you bake"
```

- Logical vector

```
myCharVec[c(TRUE,FALSE,TRUE)]
```

```
## [1] "I like pie"                "I like anything you bake"
```

Vectors - “setting”

Vectors are set in the same way, using the assignment operator: <- OR =

Vectors - “setting”

Vectors are set in the same way, using the assignment operator: <- OR =

- String vector

```
myCharVec[c(2,3)] <- c('Cats','Dogs')
```

Vectors - "setting"

Vectors are set in the same way, using the assignment operator: <- OR =

- String vector

```
myCharVec[c(2,3)] <- c('Cats','Dogs')
```

- Logical vector

```
myCharVec[c(TRUE,FALSE,FALSE)] = 'Parakeets'
```

Vectors - "setting"

Vectors are set in the same way, using the assignment operator: <- OR =

- String vector

```
myCharVec[c(2,3)] <- c('Cats','Dogs')
```

- Logical vector

```
myCharVec[c(TRUE,FALSE,FALSE)] = 'Parakeets'
```

- Results:

```
myCharVec #Here's what's inside the whole thing
```

```
## [1] "Parakeets" "Cats"      "Dogs"
```

Class conversions

Vectors (or other data) can be converted between **classes**, usually using *as.something* statements:

Class conversions

Vectors (or other data) can be converted between **classes**, usually using *as.something* statements:

- Logical to numeric

```
as.numeric(myLogVec)
```

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

Class conversions

Vectors (or other data) can be converted between **classes**, usually using *as.something* statements:

- Logical to numeric

```
as.numeric(myLogVec)
```

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

- Numeric to character

```
as.character(myNumVec)
```

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

Class conversions

Vectors (or other data) can be converted between **classes**, usually using *as.something* statements:

- Logical to numeric

```
as.numeric(myLogVec)
```

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

- Numeric to character

```
as.character(myNumVec)
```

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

- Characters to **factors**: these represent *categories* or experimental levels

```
as.factor(myCharVec) #Default order is alphabetical
```

```
## [1] Parakeets Cats      Dogs
## Levels: Cats Dogs Parakeets
```

Dataframes

- Dataframes look similar to matrices, but can hold different data types in each column:

```
myDF <- data.frame(stringCol=myCharVec, numCol=myNumVec[c(1:3)],
  logCol=myLogVec[c(1:3)])
myDF
```

```
##   stringCol numCol logCol
## 1 Parakeets      1   TRUE
## 2      Cats      2   TRUE
## 3      Dogs      3  FALSE
```

Dataframes

- Dataframes look similar to matrices, but can hold different data types in each column:

```
myDF <- data.frame(stringCol=myCharVec, numCol=myNumVec[c(1:3)],
  logCol=myLogVec[c(1:3)])
myDF
```

```
##   stringCol numCol logCol
## 1 Parakeets      1   TRUE
## 2      Cats      2   TRUE
## 3      Dogs      3  FALSE
```

-
```
summary(myDF) #This function summarizes each column
```

```
##   stringCol             numCol      logCol
##  Length:3           Min.   :1.0   Mode :logical
##  Class :character   1st Qu.:1.5   FALSE:1
##  Mode  :character   Median :2.0   TRUE :2
##                     Mean   :2.0
##                     3rd Qu.:2.5
##                     Max.   :3.0
```

Accessing Dataframes

Dataframes can be accessed numerically, by their name slots (using the $ operator), or with a mixture of the two:

Accessing Dataframes

Dataframes can be accessed numerically, by their name slots (using the $ operator), or with a mixture of the two:

- ```
  myDF[1,2]
  ```

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

Accessing Dataframes

Dataframes can be accessed numerically, by their name slots (using the $ operator), or with a mixture of the two:

- ```
  myDF[1,2]
  ```

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

- ```
  myDF$numCol #This gets all of the column "numCol"
  ```

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

Accessing Dataframes

Dataframes can be accessed numerically, by their name slots (using the $ operator), or with a mixture of the two:

- ```
  myDF[1,2]
  ```

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

- ```
  myDF$numCol #This gets all of the column "numCol"
  ```

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

- ```
  myDF[1,"numCol"]
  ```

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

Accessing Dataframes

Dataframes can be accessed numerically, by their name slots (using the $ operator), or with a mixture of the two:

- ```
  myDF[1,2]
  ```

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

- ```
  myDF$numCol #This gets all of the column "numCol"
  ```

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

- ```
  myDF[1,"numCol"]
  ```

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

- ```
  myDF$numCol[1]
  ```

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

Manipulating dataframe

Like other objects, you can alter parts of dataframes

- You can add columns

```
myDF$numCol2 <- myDF$numCol*3 #Multiplies
```

Manipulating dataframe

Like other objects, you can alter parts of dataframes

- You can add columns

```
myDF$numCol2 <- myDF$numCol*3 #Multiplies
```

- You can also alter columns in place, or elements within columns

```
myDF$numCol <- (myDF$numCol)^2
myDF$numCol[3] <- myDF$numCol[3] - myDF$numCol[2]
```

Manipulating dataframe

Like other objects, you can alter parts of dataframes

- You can add columns

```
myDF$numCol2 <- myDF$numCol*3 #Multiplies
```

- You can also alter columns in place, or elements within columns

```
myDF$numCol <- (myDF$numCol)^2
myDF$numCol[3] <- myDF$numCol[3] - myDF$numCol[2]
```

- You can delete columns by subsetting the dataframe, or assigning the column to NULL

```
myDF <- myDF[,c(1,2)] #Selects only column 1 and 2
myDF$numCol <- NULL #Removes numCol
```

Reading csv files

- One very common practice is to read in your own data from a csv file as a dataframe. Excel files can be read in directly, but present other problems.

```
testDat <- read.csv('test_results.csv') #Path to csv file
head(testDat) #head shows only the first 6 rows of dataframe
```

```
##   Concentration Treatment Lab.Member Time.of.Day
## 1           2.9   Control       Will     Morning
## 2           3.2   Control       Will     Morning
## 3           3.6   Control       Will     Morning
## 4           5.6         A       Will     Morning
## 5           6.8         A       Will     Morning
## 6           7.0         A       Will     Morning
```

Reading csv files

- One very common practice is to read in your own data from a csv file as a dataframe. Excel files can be read in directly, but present other problems.

```
testDat <- read.csv('test_results.csv') #Path to csv file
head(testDat) #head shows only the first 6 rows of dataframe
```

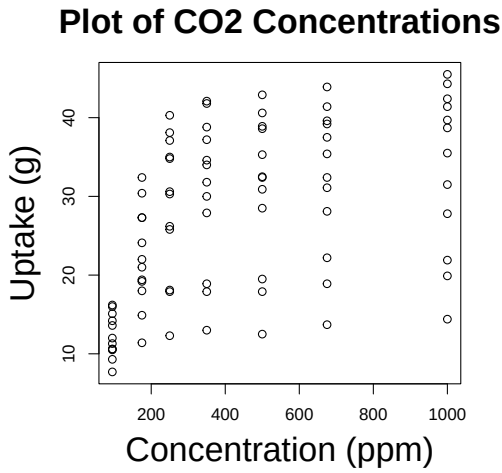
```
##   Concentration Treatment Lab.Member Time.of.Day
## 1           2.9   Control       Will     Morning
## 2           3.2   Control       Will     Morning
## 3           3.6   Control       Will     Morning
## 4           5.6         A       Will     Morning
## 5           6.8         A       Will     Morning
## 6           7.0         A       Will     Morning
```

- R can't handle spaces or other special characters in the column headers (replaces them with periods). It also tries to guess the proper data type for each column, but sometimes gets this wrong.

Plotting

- The *plot* command is useful for quickly looking at sets of data. The following CO2 dataset is built-in to R.[1]

```
#Makes a plot of the uptake (y) and
# concentration (x) columns of CO2
# dataframe, and customizes axis labels
plot(x = CO2$conc, y = CO2$uptake,
     xlab = 'Concentration (ppm)',
     ylab = 'Uptake (g)',
     main = 'Plot of CO2 Concentrations',
     cex.lab = 2, cex.main=2)
```



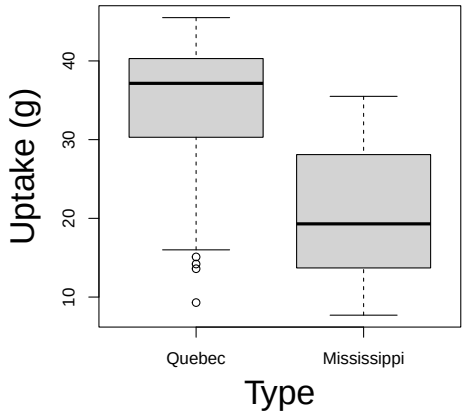
[1] To see others, type `data()` in the console

More Plotting

- The boxplot command can summarize *continuous* and *categorical* data

```
#Boxplot uses a formula rather than x,y
#   vectors

#Formula: uptake depends on (~) Type
boxplot(CO2$uptake ~ CO2$Type,
        xlab='Type',
        ylab='Uptake (g)',
        cex.lab = 2, cex.main=2)
```



First challenge

Your supervisor has just given you a dataset (*test_results.csv*) recorded by two undergrads. However, these undergrads were in a hurry and have made some mistakes:

- Make a **script** in R, and use this to record what you do
- Read the *csv* file and fix any mistakes. Bonus if you do this without using Excel!
- Plot the concentration data by treatment group, then plot it for each undergrad. Does there look like much of a difference?
- Some useful commands: **read.csv**, **boxplot**, **is.na**, **as.factor**, **summary**

Functions

- Functions take objects as **arguments** (input) and return other **objects** (output)

```
myNumVec <- c(1,2,3,4,5)
meanVec <- mean(myNumVec) #Arithmetic mean (average)
sdVec <- sd(myNumVec) #Standard deviation (sqrt(variance))
meanSdVec <- c(meanVec,sdVec) #Joins mean and SD into a vector
meanSdVec
```

```
## [1] 3.000000 1.581139
```

Functions

- Functions take objects as **arguments** (input) and return other **objects** (output)

```
myNumVec <- c(1,2,3,4,5)
meanVec <- mean(myNumVec) #Arithmetic mean (average)
sdVec <- sd(myNumVec) #Standard deviation (sqrt(variance))
meanSdVec <- c(meanVec,sdVec) #Joins mean and SD into a vector
meanSdVec
```

```
## [1] 3.000000 1.581139
```

- If you can't remember how a command works, use ? to access the help files

```
?median
```

Homemade Functions

- You can make your own functions! This is useful if you have to do the same thing to many different input objects.

```
myFun <- function(input){ #Takes a vector of numbers
  A <- mean(input) #Take the mean of INPUT
  B <- sd(input) #Take the SD of INPUT
  C <- c(A,B) #Join A and B into a vector C
  return(C) #Return (output) C, then end the function
}
myFun(myNumVec) #Same as previous slide
```

```
## [1] 3.000000 1.581139
```

Homemade Functions

- You can make your own functions! This is useful if you have to do the same thing to many different input objects.

```
myFun <- function(input){ #Takes a vector of numbers
  A <- mean(input) #Take the mean of INPUT
  B <- sd(input) #Take the SD of INPUT
  C <- c(A,B) #Join A and B into a vector C
  return(C) #Return (output) C, then end the function
}
myFun(myNumVec) #Same as previous slide
```

```
## [1] 3.000000 1.581139
```

- The objects inside of functions (A, B, C in the one above) disappear after the function runs. However, *functions can see objects in the outer environment*, so beware of the Steve Problem*

Summary statistics

Often we want to get the mean of one columns, but split it up by other things in the dataframe. Using the CO2 plant example, how does *uptake* differ between *Type*?

Summary statistics

Often we want to get the mean of one columns, but split it up by other things in the dataframe. Using the CO2 plant example, how does *uptake* differ between *Type*?

```
#Split up uptake by Type and Treatment,
#   then take the mean
tapply(CO2$uptake, list(CO2$Type,
                CO2$Treatment), mean)
```

```
##             nonchilled  chilled
## Quebec        35.33333 31.75238
## Mississippi   25.95238 15.81429
```

Summary statistics

Often we want to get the mean of one columns, but split it up by other things in the dataframe. Using the CO2 plant example, how does *uptake* differ between *Type*?

```
#Split up uptake by Type and Treatment,
#   then take the mean
tapply(CO2$uptake, list(CO2$Type,
                CO2$Treatment), mean)
```

```
##             nonchilled  chilled
## Quebec        35.33333 31.75238
## Mississippi   25.95238 15.81429
```

Typing "CO2" over and over again is annoying. You can use *with* to avoid this (avoid using *attach*):

Summary statistics

Often we want to get the mean of one columns, but split it up by other things in the dataframe. Using the CO2 plant example, how does *uptake* differ between *Type*?

```
#Split up uptake by Type and Treatment,
#   then take the mean
tapply(CO2$uptake, list(CO2$Type,
                CO2$Treatment), mean)
```

```
##             nonchilled  chilled
## Quebec         35.33333 31.75238
## Mississippi    25.95238 15.81429
```

Typing "CO2" over and over again is annoying. You can use *with* to avoid this (avoid using *attach*):

```
#Runs command inside the name space of the
#   CO2 object
with(CO2,
 tapply(uptake, list(Type, Treatment), sd)
)
```

```
##             nonchilled  chilled
## Quebec         9.596371 9.644823
## Mississippi    7.402136 4.058976
```

if statements

- R can be told to do things only *if* certain conditions apply. This is useful inside of functions for error handling:

```
myFun2 <- function(x){
  xClass <- class(x) #What class is x? (Numeric, character, boolean)

  if(xClass=='character'){ #== means "are these things equal"?
    return('This is a string') #If x is a character, returns a message
  } else {
    return(mean(x)) #If x isn't a character, returns the mean of x
  }
}
myFun2(myCharVec)
```

```
## [1] "This is a string"
```

```
myFun2(myNumVec)
```

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

for loops

- R can be told to do things repeatedly, using an *index* inside a loop:

```
classVec <- rep(0,10) #Storage vector of zeros, 10 long
classVec[c(1,2)] <- 1 #Set first two slots to 1

#Each time the loop repeats, i will take on values 3 to 10
for(i in 3:length(classVec)){

  #ith slot of classVec becomes the sum of the previous two slots
  classVec[i] <- classVec[i-1] + classVec[i-2]
}
classVec #First 10 numbers in the Fibbonaci sequence
```

```
##  [1]  1  1  2  3  5  8 13 21 34 55
```

Second challenge

Population growth models are common in ecology, and usually often take the form $n_t = n_{t-1} + rn_{t-1}$, where n is the number of critters at some time point t, and r is the change in n from one point to the next (r = 0: no change). Write a `function` (with a `for` loop inside) that performs a simple population simulation using the following models:

- Exponential growth: $n_t = n_{t-1}(1 + r)$
- Logistic growth: $n_t = n_{t-1}(1 + r(1 - \frac{n_{t-1}}{k}))$

Hint: functions need input variables that tell them what to do. The input variables here could be things like *starting population* (n_0), *growth rate* (r), and *number of time steps* (T) to simulate