



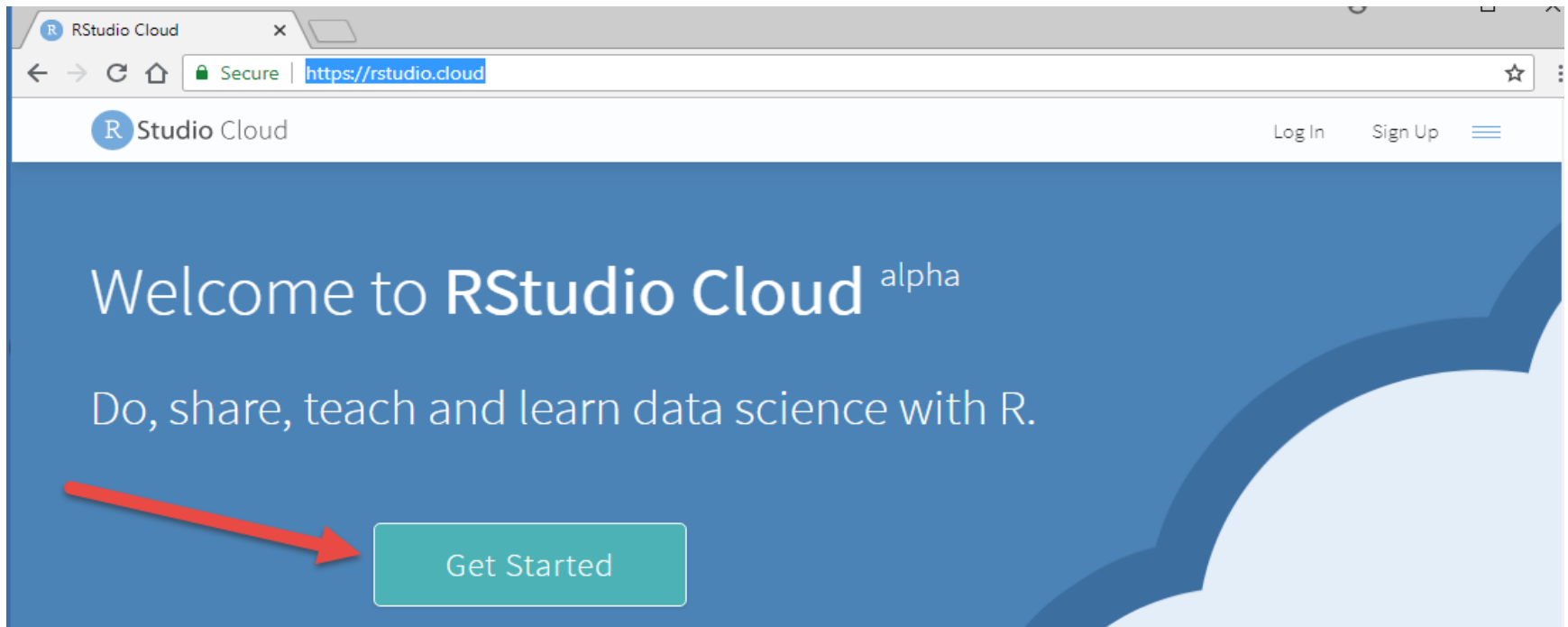
# R OVERVIEW

HARVARD EXTENSION SCHOOL

# RSTUDIO CLOUD

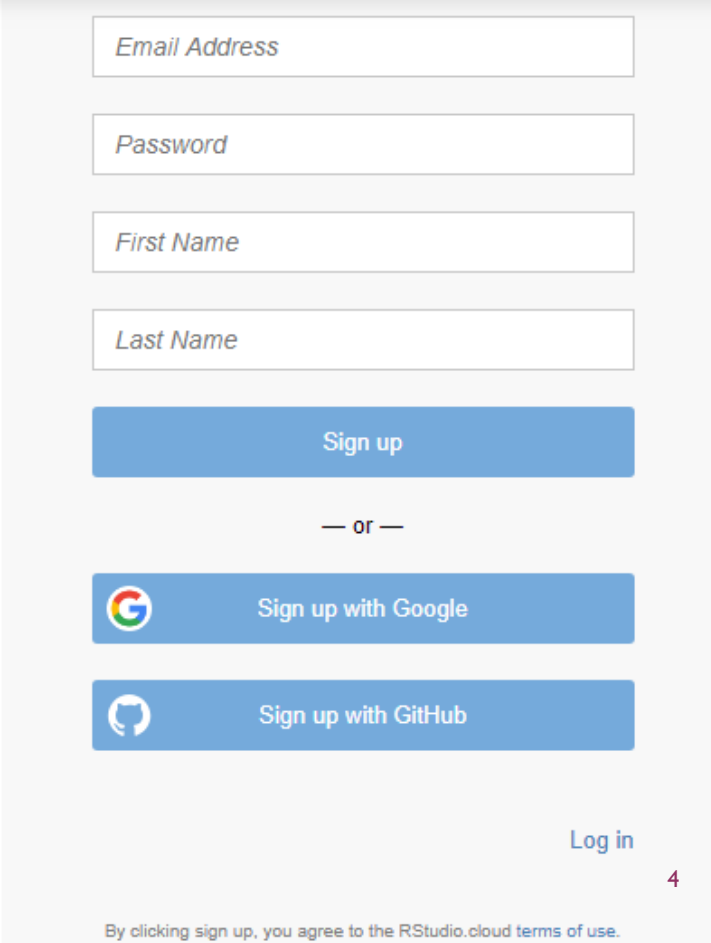
- We use RStudio Cloud because:
  - you don't need to have your own powerful computer with administrative rights (can do your homework anywhere, even on a tablet or Chromebook).
  - RStudio Cloud is powerful enough (enough memory and disk space).
  - You will have the same version of your libraries.
  - You will not need individual support since everything is the same (same errors for everyone, and not depending on the version of your system).

# RSTUDIO CLOUD



# CREATE A LOG-IN

- Sign-up or Log in if you already have an account
- Note: You can use your google or GitHub account if you have one



The image shows a login and sign-up form for RStudio Cloud. It features four input fields for 'Email Address', 'Password', 'First Name', and 'Last Name'. Below these is a blue 'Sign up' button. A separator '— or —' is followed by two social login buttons: 'Sign up with Google' (with the Google logo) and 'Sign up with GitHub' (with the GitHub logo). At the bottom right is a 'Log in' link. A footer note states: 'By clicking sign up, you agree to the RStudio.cloud terms of use.'

Email Address


Password


First Name

Last Name

Sign up

— or —

 Sign up with Google

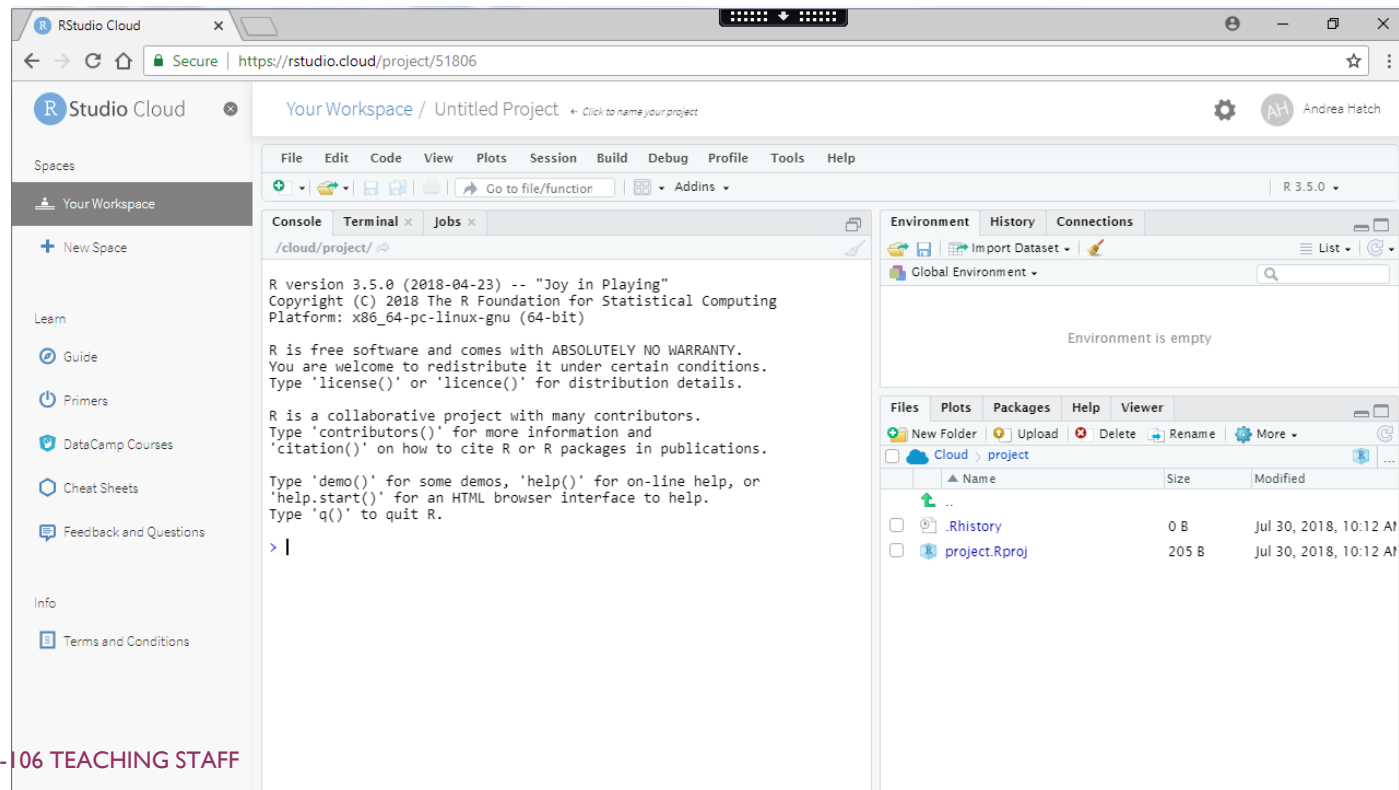
 Sign up with GitHub

[Log in](#)

By clicking sign up, you agree to the RStudio.cloud [terms of use](#).

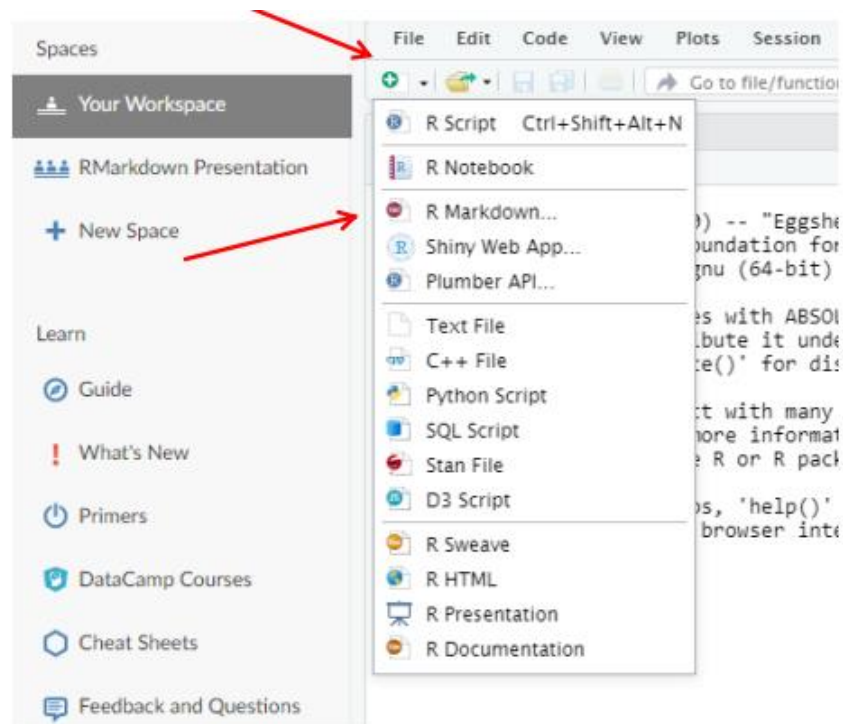
# GET STARTED

- To get started select New Project and you are ready to go



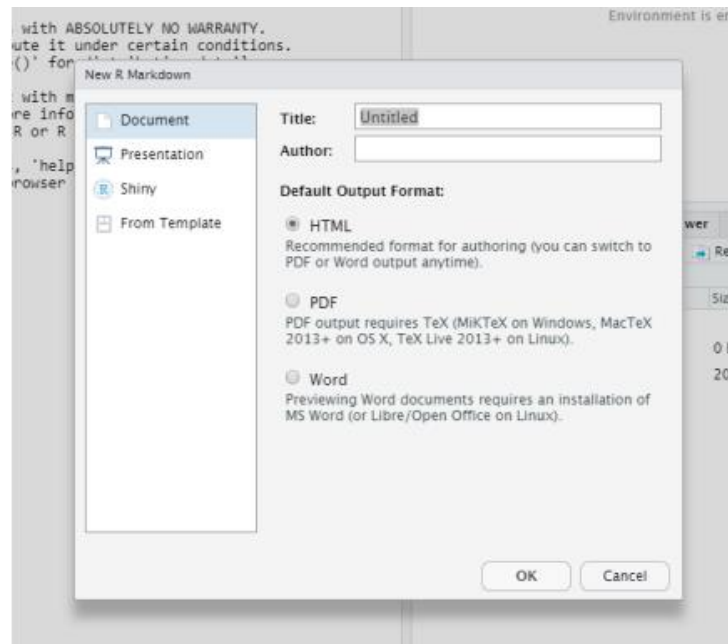
# GET STARTED WITH R MARKDOWN

- Click the plus button and select R Markdown (this is what is required to be turned in for your homework)



# GET STARTED WITH R MARKDOWN - 2

- You will need to update packages (click “Yes” when prompted)
- Choose title, author name and what is the output



# GET STARTED WITH R MARKDOWN - 3

- R Markdown uses so called “literate programming” that mixes plain text and output of executed code
- R Markdown also promotes reproducibility because what you don’t include in the code that won’t be executed (not like Excel)
  1. You can write regular text
  2. You can write in LaTeX notation:
    - `$$ \beta = (X^T X)^{-1} X^T y $$` translate to  $\beta = (X^T X)^{-1} X^T y$
  3. R code inline (``r 1+1``)
  4. R code in so-called chunks (starting ```{r}` ; ending ````)
    - RMarkdown can execute also Python, C++ (RCpp), SQL and others



# GET STARTED WITH R MARKDOWN - 4

- See Rmarkdown web page: <https://rmarkdown.rstudio.com/> ; esp. Gallery
- See the RMarkdown cheat sheet:  
<https://www.rstudio.com/resources/cheatsheets/>

# TEXTBOOKS AND OTHER RESOURCES FOR LEARNING R

- Paul Gerrard & Radia M. Johnson, Mastering Scientific Computing with R
- Julian Faraway, Linear Models with R
- P. Kuhnert & B. Venables, An Introduction to R: Software for Statistical Modeling & Computing
- Adler, J: R in a Nutshell
- Teetor, P.: R Cookbook
- Check <https://cran.r-project.org/> for manuals and other resources/
- If you are an R beginner (no endorsement), we've heard videos from R programming course from JHU/Coursera is good (can be found on youtube)

# R AS A CALCULATOR

- `> 1 + 1 # Simple Arithmetic`
  - `[1] 2`
- `> 2 + 3 * 4 # Operator precedence`
  - `[1] 14`
- `> 3 ^ 2 # Exponentiation`
  - `[1] 9`
- `> exp(1) # Basic mathematical functions are available`
  - `[1] 2.718282`
- `> sqrt(10)`
  - `[1] 3.162278`
- `> pi # The constant pi is predefined`
  - `[1] 3.141593`

# R INTRODUCTIONS

- Results of calculations can be stored in objects using the assignment operators:
  - An arrow (`<-`) formed by a smaller than character and a hyphen without a space!
  - The equal character (`=`).
  - `X=2`
  - `Y<-2`
- These objects can then be used in other calculations. To print the object just enter the name of the object. There are some restrictions when giving an object a name:
  - Object names cannot contain 'strange' symbols like `!`, `+`, `-`, `#`.
  - A dot (`.`) and an underscore (`_`) are allowed, also a name starting with a dot.
  - Object names can contain a number but cannot start with a number.
  - R is case sensitive, `X` and `x` are two different objects, as well as `DAT` and `daT`.

## R INTRODUCTIONS, *CONT'D*

- To list the objects that you have in your current R session use the function `ls` or the function `objects`.
  - `> ls()`
  - `[1] "x" "y" "Y" "z"`
- If you assign a value to an object that already exists then the contents of the object will be overwritten with the new value (without a warning!).
  - `X<-2`
  - `X<-3`
- Use the function `rm` to remove one or more objects from your session.
  - `> rm(x) #trick to remove everything: rm(list=ls())`

# R INTRODUCTIONS, *CONT'D*

- The collection of objects that you currently have is called the workspace and it is not automatically saved by R. Always save your workspace !
- R gets confused if you use a path in your code like  
`c:\mydocuments\myfile.txt`
- This is because R sees "\" as an escape character. Instead, use  
`c:\\my documents\\myfile.txt`
  - or  
`c:/mydocuments/myfile.txt`
- Once R is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:  
`help.start()`    # general help  
`help(ls)`        # help about function ls  
`?ls`            # same thing  
`apropos("ls")` # list all function containing string ls  
`example(ls)`    # show an example of function ls

# R INTRODUCTIONS, *CONT'D*

- R comes with a number of sample datasets that you can experiment with.
- Type `data()` to see the available datasets. The results will depend on which packages you have loaded.
  - `> data()`

Data sets in package 'datasets':

AirPassengers	Monthly Airline Passenger Numbers 1949-1960
BJsales	Sales Data with Leading Indicator
BJsales.lead (BJsales)	Sales Data with Leading Indicator
BOD	Biochemical Oxygen Demand
CO2	Carbon Dioxide Uptake in Grass Plants
ChickWeight	Weight versus age of chicks on different diets

- Type `help(datasetname)` for details on a sample dataset.
  - `> help(AirPassengers)`

## R INTRODUCTIONS, *CONT'D*

- R is an open source. You can write new functions and package those functions in a so called 'R package' (or 'R library') or you can use R packages written by others.
- There is a lively R user community and many R packages have been written and made available on CRAN for other users.
- `library()` # to see what R packages you have
- `install.packages("MASS")` # install MASS package
- `install.packages("faraway")` # install faraway package
- `library("faraway")` # to load a library



# R VECTORS

- A series of numbers
- Created with
  - `c()` to concatenate elements or sub-vectors
  - `rep()` to repeat elements or patterns
  - `seq()` or `m:n` to generate sequences
- Examples:
  - `x <- c(1, 2, 3)`  
[1] 1 2 3
  - `y <- seq(1:3)`  
[1] 1 2 3
  - `z <- rep(3, 14)`  
[1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3

# DEFINING VECTORS

- `> rep(1,10) # repeats the number 1, 10 times`
  - `[1] 1 1 1 1 1 1 1 1 1 1`
- `> seq(2,6) # sequence of integers between 2 and 6`
  - `[1] 2 3 4 5 6 # equivalent to 2:6`
- `> seq(4,20,by=4) # Every 4th integer between 4 and 20`
  - `[1] 4 8 12 16 20`
- `> x <- c(2,0,0,4) # Creates vector with elements 2,0,0,4`
- `> y <- c(1,9,9,9)`
- `> x + y # Sums elements of two vectors`
  - `[1] 3 9 9 13`
- `> x * 4 # Multiplies elements`
  - `[1] 8 0 0 16`
- `> sqrt(x) # Function applies to each element`
  - `[1] 1.41 0.00 0.00 2.00 # Returns vector`

# OPERATIONS ON VECTORS

## Arithmetic operators

$+$   $x$

$-$   $x$

$x + y$

$x - y$

$x * y$

$x / y$

$x ^ y$

$x \% y$

$x \%/ y$

# OPERATIONS ON VECTORS: EXAMPLE

- For example, if we multiply a vector by 2, all the elements of the vector will be multiplied by 2
  - `> x <- c(1, 3, 5, 10) > x * 2`
  - `[1] 2 6 10 20`
- `> x <- c(1, 3, 5, 10) ; y <- c(13, 15, 17, 22) ; x + y`
  - `[1] 14 18 22 32`
- If the vectors are of different lengths, the shorter vector will be extended to match the length of the longer vector by recycling its elements starting from the first element. However, you will also get a warning message from R in case you did not intend to add vectors of differing length, as follows:
  - `z <- c(1,3, 4, 6, 10) ; x + z #1 was recycled to complete the operation.`
    - `[1] 2 6 9 16 11 Warning message: In x + z : longer object length is not a multiple of shorter object length`
  - the standard operators also have `%%`, which indicates  $x \bmod y$ , and `/%`, which indicates integer division as follows:
    - `> x %% 2`
      - `[1] 1 1 1 0`
    - `> x %/% 5`
      - `[1] 0 0 1 2`

# LISTS

- A list is a generalization of a vector and represents a collection of data objects. A list allows you to gather a variety of (possibly unrelated) objects under one name. Here are some examples:

- `hd<- list(name="John Smith", id=1111, grade="B+", age=35)`

- `> hd`

- `$name`
- `[1] "John Smith"`
- `$id`
- `[1] 1111`
- `$grade`
- `[1] "B+"`
- `$age`
- `[1] 35`

Four levels: name, id, grade, and age

- Identify elements of a list using the `[[]]` convention.
- `hd[[2]] # 2nd component of the list`
- `[1] 1111`

# FACTORS

- Categorical data can also be stored as Factors. These are specialized vectors that contain predefined values referred to as Levels.
- For example, say you have data for "placebo" and "treatment" for four patients, you could store this information as factors instead of a character vector by using the following code:
- `drug_response <- c("placebo", "treatment", "placebo", "treatment")`
- `levels(drug_response)`
  - `NULL` # R did not store drug\_response as a factor.
- `drug_response <- factor(drug_response)` #factor function or `as.factor` function will store it as a factor
- `levels(drug_response)`
  - `[1] "placebo" "treatment"`
- To check the integers used for each level, you can use the `as.integer()` function as follows:
- `> as.integer(drug_response) [1] 1 2 1 2`

# MATRICES

- All columns in a matrix must have the same mode(numeric, character, etc.) and the same length. The general format is
- `mymatrix <- matrix(vector, nrow=r, ncol=c, byrow=FALSE, dimnames=list(char_vector_rownames, char_vector_colnames))`
  - `byrow=TRUE` indicates that the matrix should be filled by rows.
  - `byrow=FALSE` indicates that the matrix should be filled by columns (the default).
  - `dimnames` provides optional labels for the columns and rows.
- Example: what are the differences between two matrices?
  - `y<-matrix(1:20,byrow=F, nrow=5,ncol=4)`
  - `y<-matrix(1:20,byrow=T, nrow=5,ncol=4)`

# MULTIDIMENSIONAL ARRAYS

- Arrays are similar to matrices but can have more than two dimensions. See **help(array)** for details.
- `list_example = list((1:2), (1:5))`
- `[[1]]`
- `[1] 1 2 3`
- `[[2]]`
- `[1] 1 2 3 4 5`



# DATA FRAME

- The most common way to store data in R is through data frames and it makes data analysis much easier, especially when dealing with categorical data.
- Data frames are similar to matrices, except that each column can store different types of data.
- You can construct data frames using the `data.frame()` function or convert an R object into a data frame using the `as.data.frame()` function as follows:
  - `students <- c("John", "Mary", "Ethan", "Dora")`
  - `score <- c(76, 82, 84, 67)`
  - `grade <- c("B", "A", "A", "C")`
  - `class <- data.frame(students, score, grade)`
- ```
> class
```
- ```
  students score grade
```
- ```
1     John    76     B
```
- ```
2     Mary    82     A
```
- ```
3     Ethan    84     A
```
- ```
4     Dora    67     C
```

# USEFUL FUNCTIONS

```
length(object) # number of elements or components
str(object)    # structure of an object
class(object)  # class or type of an object
names(object)  # names
c(object1,object2,...) # combine objects into a vector
cbind(object,1 object2, ...) # combine objects as columns
rbind(object,1 object,2 ...) # combine objects as rows
ls()           # list current objects
rm(object)     # delete an object
newobject <- edit(object) # edit copy and save a
newobject
fix(object)
```

# LOADING DATA INTO R

- You can see where R will read and save files, by default, using the `getwd()` function. Then, you can change it using the `setwd()` function
- `getwd()`
  - `[1] "/cloud/project"`
- Use `read.table()`, `read.delim()`, `read.csv()` functions to load the delimited data into R
  - `myData.df <- read.table("myData.txt", header=TRUE, sep="\t")`
  - `read.delim("myData.txt", header=TRUE)`
  - `myData2.df <- read.csv("myData.csv", header=FALSE)`
- To load excel files first install `gdata` package and use `read.xls` function
  - `install.packages("gdata")`
  - `myData.df <- read.xls("myData.xlsx", sheet=1)` #also uploads .xls files and returns a data frame

# SAVING / EXPORTING FILES

- To save an object, preferably a matrix or data frame, you can write a .txt or .csv file or a file using another delimiter using the `write.table()` or `write.csv()` functions.
- You can choose to include `row.names` and `col.names` by setting these arguments to `TRUE`.
- The output file will be saved to your current directory.
  - `write.table(myData.df, file="savedata_file.txt", quote = FALSE, sep= "\t", row.names=TRUE, col.names=NA, append=FALSE)`
  - `write.csv(myData.df, file = "savedata_file.csv")` #same output as above

# MISSING DATA

- In **R**, missing values are represented by the symbol **NA** (not available) .
  - `y <- c(1,2,3,NA)`
  - `is.na(y)` # returns a vector (F F F T)
  - `na.omit(y)` # delete missing values
- Arithmetic functions on missing values yield missing values.
  - `mean(y)`
  - `[1] NA`
  - `mean(y, na.rm=TRUE)` # na.rm removes the missing values
  - `[1] 2`

# LOGICAL OPERATORS

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	Not x
x   y	x OR y
x & y	x AND y
isTRUE(x)	test if x is TRUE

# FLOW CONTROL

- **for**
  - `for (var in seq) {expr}`
- **if-else**
  - `if (cond) {expr}`  
`if (cond) {expr1} else {expr2}`
- **ifelse**
  - `ifelse(test,yes,no)`
- **while**
  - `while (cond) {expr}`
- **switch**
  - `switch(expr, ...)`

# THE FOR() LOOP

- The `for(i in vector){commands}` statement allows you to repeat the code written in brackets `{}` for each element (i) in your vector in parenthesis.
  - `for(i in 1:5){x[i] <- x[i+2] + x[i+1]}`
- Example: Write for loop function that calculates the cumulative total for each row or element



# IF() AND ELSE IF () STATEMENTS

- The `if(condition){commands}` statement allows you to evaluate a condition and if it returns TRUE, the code in brackets will be executed.
- You can add an `else {commands}` statement to your `if()` statement if you would like to execute a block of code if your condition returns FALSE:
  - `x <- 4 > # we indent our code to make it more legible`
  - `if(x < 10) { x <-x+4; print(x) }`
  - `[1] 8`
- If you have several conditions to test before running an `else {}` statement, you can use an `else if(condition){commands}` statement as follows:
  - `x <- 1`
  - `if(x == 2) { x <- x+4; print("X is equal to 2, so I added 4 to it.") } else if (x > 2) { print("X is greater than 2, so I did nothing to it.") } else { x <- x -4 ;print("X is not greater than or equal to 2, so I subtracted 4 from it.") }`
  - `[1] "X is not greater than or equal to 2, so I subtracted 4 from it."`

# WHILE() LOOP

- The `while(condition){commands}` statement allows you to repeat a block of code until the condition in the parenthesis returns FALSE.
- `while(count < 15) {count <- count +1;.....}` when count is greater than 15, loop will stop.
- You can also use repeat and break statement, please refer to the r-help or manual.

# SORTING

- To sort a dataframe in R, use the `order( )` function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.
- ```
data(mtcars)
# sort by mpg
newdata = mtcars[order(mtcars$mpg),]
# sort by mpg and cyl
newdata <- mtcars[order(mtcars$mpg, mtcars$cyl),]
# sort by mpg (ascending) and cyl (descending)
newdata <- mtcars[order(mtcars$mpg, -mtcars$cyl),]
```

# MERGING

- To merge two dataframes (datasets) horizontally, use the merge function. In most cases, you join two dataframes by one or more common key variables (i.e., an inner join).
- # merge two dataframes by ID  
total <- merge(dataframeA,dataframeB,by="ID")
- # merge two dataframes by ID and Country  
total <- merge(dataframeA,dataframeB,by=c("ID","Country"))

# MERGING – ADDING ROWS

- To join two dataframes (datasets) vertically, use the `rbind` function. The two dataframes must have the same variables, but they do not have to be in the same order.
- `total <- rbind(dataframeA, dataframeB)`
  - If dataframeA has variables that dataframeB does not, then either:
    - Delete the extra variables in dataframeA or
    - Create the additional variables in dataframeB and set them to NA (missing)
  - before joining them with `rbind`.

# WRITING R FUNCTIONS

- Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.
- `f <- function() { ## Do something interesting }`
- Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly, Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function
- The return value of a function is the last expression in the function body to be evaluated.

# MODERN R

- Tidyverse (Universe of packages around tidy data concept)
- dplyr, tibble, purrr
- Visualization using ggplot2
- RMarkdown and automated report development
- Package development using modern techniques (automated testing - testthat, computer-aided documentation – ROxygen2)
- Shiny app development (see shiny.rstudio.com), interactive
- See how Financial Times uses R for data journalism (recommended):  
<http://johnburnmurdoch.github.io/slides/r-ggplot/>

# EXAMPLE

- Write a R functions that generates a random sample 10000 observations from the normal distribution and calculates 99% percentile.
- How does everyone feel about R? Questions?