**An Introduction to R for Data Analysis**

**G. Miller and R. van Woesik**

**Department of Ocean Engineering and Marine Science,**

**Florida Institute of Technology**

**Objective:** Prepare incoming graduate students for thesis and dissertation research using the statistical software package R.

**Schedule:** MTWRF 9:00-12:00 and 1:00-4:00

Each section: 20 min lecture, 30 min working in R, 10 minutes troubleshooting, followed by a short break

**Overview:**

**Day 1** (1) Why use R? (2) Why use 4 panels in R Studio? (3) Scripts (4) Data and dataframes (5) Variables (6) Reading external data (7) Producing outputs.

**Day 2** (8) Functions (9) Transformations (10) Combining columns and rows (11) Vectors and matrices (12) Variable types.

**Day 3** (13) Subsetting (14) Sorting (15) Packages and libraries, (16) Summary tables (17)Data manipulation.

**Day 4** (18) Scatterplots (19) Plot lines (20)Graph texts (21)Graph axes (22)Histograms (23) Boxplots

**Day 5** (24) Bar plots (25) Figure sizes (26) Saving graphs (27) Vignettes (28) Pipes.

**Note to students:** R commands in the text will be in **bold font**. You must substitute the appropriate object, file path, etc. for anything in ***italics***. Special characters and shortcut key combinations will also appear in **bold font**. Example scripts will be highlighted in gray.

**Day 1**

**Morning**

**Section 1: Why use R?** 

**Rationale:** We highly recommend that you become familiar with R because you will be able to use it as a data-analysis platform for all your classes and research.

During your tenure at the university, you will be tasked, in all courses and research endeavors, to compile and analyze data. After all, science is about testing ‘testable’ hypotheses, which amounts to coming up with a useful and hopefully intriguing research question, which can be rephrased as a testable hypothesis, followed by gathering data and running statistical tests to assess the hypotheses. The best way to do the latter part is to use the statistical program R.

Modern scientific studies use R because the code that is used to analyze the data can be made available to other scientists. In other words, the code is reproducible for other studies. In this way, the code can be re-tested in the future, modified, and improved. R is free and easily accessible, has excellent graphing capacity, and is maintained by a group of experts. Firstly, you need to download and install R and download and install Rstudio. Please follow the instructions below.

**Downloading R**

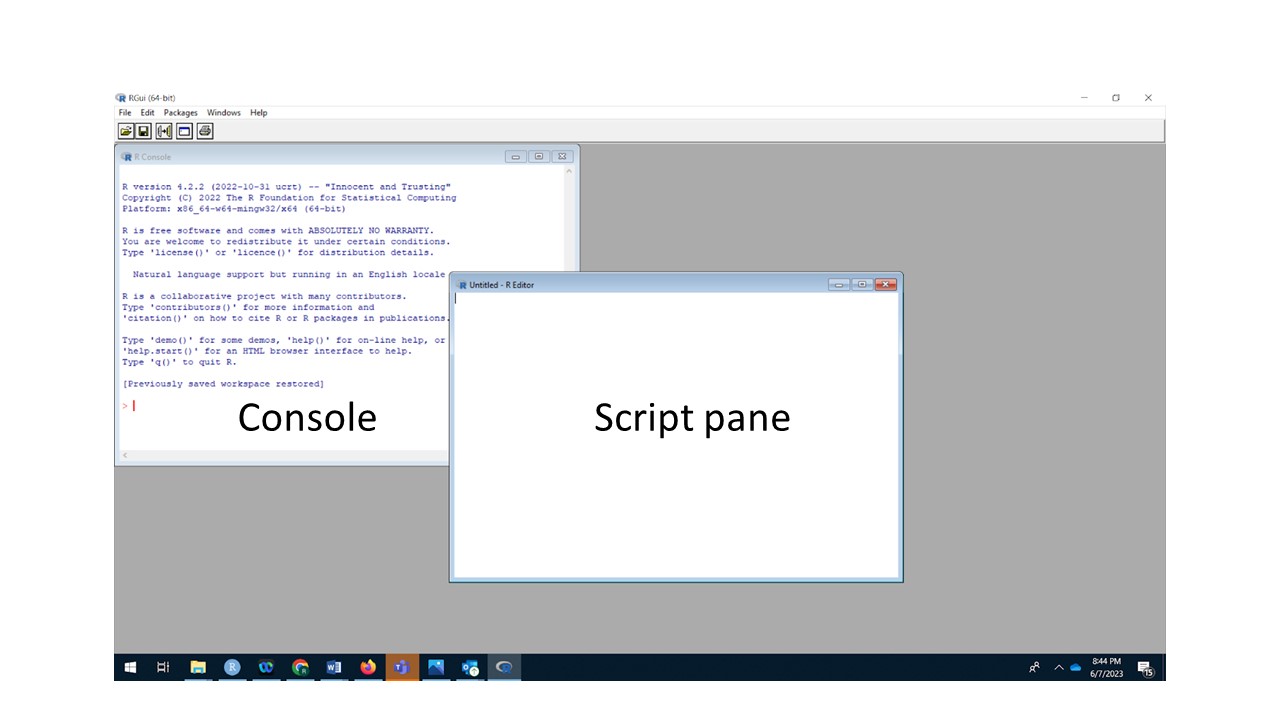
1. Go to the R Projects website, which can be found by searching for “R” in a web browser: [www.r-project.org](about:blank)
2. Click on “download R”
3. Select the CRAN Mirror that is located closest to you. It is recommended to use “0-Cloud” as it automatically redirects you to servers worldwide.
4. Click on Download “R for (your operating system)”
   1. Windows: Click on “install R for the first time”, then “Download R (version #) for Windows”
   2. Mac: Select the latest release and click on the link
      1. You will also need to install XQuartz. A link is found on the same webpage from which you downloaded R.
      2. You will need to start XQuartz for every R session. Start XQuartz before starting R or RStudio.
   3. Linux: Select your distribution and follow the instructions
   4. Chromebook: Not supported directly by the R Core Team. Search for “install R on Chromebook” and find a reliable source
5. Windows and Mac: install R like any other application

**Updating R**

1. Repeat the above instructions to download the new version of R
2. Optional: Uninstall the previous version of R
3. Optional: Find the “win-library” folder for the uninstalled version of R and delete it
   1. The command **.libPaths()** will display the folder pathway(s) to your library folder(s)
4. If you have multiple versions of R installed, RStudio (below) will either:
   1. Default to the latest version of R
   2. Ask you which version you want to associate with RStudio

**The R interface**

When you open up R, you will notice one white pane, called Console. This is where all commands are executed. Go to ‘File’ and click on ‘New Script’, and a second pane will appear. This second panel is the Script pane, where you write code that can be saved, just as you save and edit a Word document. You can also “Open script” to open an existing script file. A script can be 1 or 2 lines long or several thousand lines long. **Scripts are important and should always be saved, frequently, during an editing session.** Scripts are written in the Script pane and executed in the Console. Note that the R interface is a no-frills way to execute R scripts and you can use it exclusively without any graphical interfaces, such as RStudio. The graphic window will pop up if you create a graph.



Console on the left and Script pane on the right.

**A more user-friendly approach, using RStudio**

RStudio is an alternate interface for R. Remember that R is the driving engine for all your analysis, and RStudio is the fancy facade that allows you to drive the engine efficiently. It is developed by Posit (https://posit.co/download/rstudio-desktop/. You can also download base R and RStudio from the Posit website. You must install R on your computer before installing RStudio but once RStudio is installed you can use it to run R without starting R separately.

**Downloading RStudio**

1. Go to the Posit website, which can be found by searching for “RStudio” in a web browser: [posit.co](about:blank)
2. Click on “Download RStudio”
3. Below the heading “RStudio Desktop” click on “Download RStudio”
4. Scroll down, if needed, to find your operating system or Linux distribution
5. Click on the file name to download
6. Install RStudio like any other application

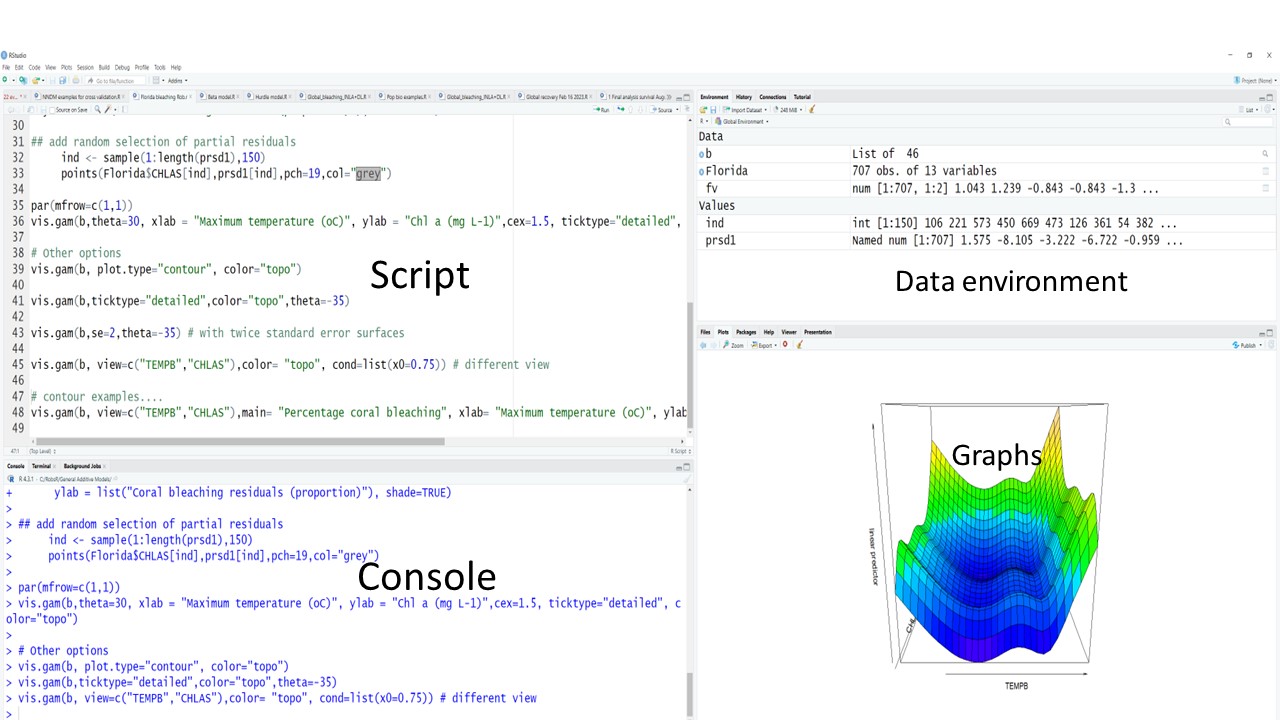
**Updating RStudio**

1. Open RStudio
2. In the “Help” menu, click on “Check for Updates”
3. If updates are available, click on “Quit and Download…”. This will open the RStudio download site in your default web browser.
4. Scroll down, if needed, to find your operating system or Linux distribution
5. Click on the file name to download
6. Install RStudio like any other application
7. The new version of RStudio will install over the old version; no need to uninstall anything

**Task:** Place R (4.3.1) and R\_Studio as icons on your Desktop.

**Section 2: Why use 4 panes in R Studio?**

**Rationale:** We need to understand the layout and how to navigate the R software environment.

****

R Studio allows the user (i.e., you) to write scripts, run scripts, generate graphs, visualize graphical outputs, and examine all your data simultaneously. This is why R users use R Studio because it is extremely practical to visualize all 4 panes (i.e., script, console, graphical output, and data environment) at the same time and place. In this way, you can see where you may have a problem and how your graph can be improved. Sure, in base R, without R Studio, users can write a script (i.e., a chunk of code) and then run the script in the Console, and then visualize the output, and then go back to the script, however, seeing all 4 panes simultaneously has several advantages.

If you don’t like the 4 (default) panes you can reduce them to 2. These panes can also be resized, maximized, or minimized, and we can also change the color, change font, and font size depending on your taste and vision. To make a change to the panes, select the “Tools” menu then “Global Options” and then “Pane Layout”. These tabs correspond to the R panes detailed above.

Using R Studio, you will still need to open the script editor, called “Source” in RStudio, and can do so under “File”, then “New File”, then “R Script” (for new script) or “File”, then “Open File” (to open an existing script file). Here are some more details on each of the four main R Studio features.

**The Console** is the non-graphical output displayed here, the console can be used to enter commands but this is not recommended to enter commands here because they will be lost when you finish your R session.

**The Script** is the best place to enter, edit, and run computer code. The script editor will need to be opened when you start R or RStudio. Scripts are lines containing executable commands and comments. Scripts can and should be saved as .R files.

**The graphical window** is where plots are displayed.

**The Data (Workspace) Environment** is where data, variables, functions, etc; these components are collectively called objects. Every object must have a unique name that begins with a letter. R is case sensitive so “g” and “G” are different objects as are “amm”, “Aam”, “aAm”, “aaM”, “AAm”, and so forth. Object names can contain numbers but must begin with a letter. Object names can contain periods but not spaces or other symbols. The workspace can be saved as R data files.

**Task:** Experiment on your computer with the 4 panels: change their color and font size to suit your preference.

**Keeping track of your scripts and data is vital**. Think about a useful organizing strategy to save your scripts. You may like to use keywords, or project names, such as “Bleaching\_the winners and losers”, and then place all the data and scripts for each project within that folder. **It is useful to quickly get into a habit of organizing your files in folders so that you can find the files quickly.** Remember, during your graduate school tenure you will end up with potentially thousands of files and hundreds of folders for data analysis. These folders will be recognized by R as working directories. R needs to be told where your data is located: for example,

C:/Marks\_R/Bleaching\_the\_winners\_and \_losers. This folder will be the working directory for any analysis on coral bleaching on Mark’s computer.

**The working directory.** A folder on the hard drive of your computer or other storage media where R will default to when asked to read or save a file. The command **getwd()** will display the file path of the working directory in the console. To change the working directory either:

* + - **setwd(“*file path”*)** will change the working directory. Use “/” instead of “\” when specifying the file path
    - In the “Session” menu select “Set Working Directory” and “Choose Directory” then navigate to the desired folder and click “Open”

**Task:** Outline a simple naming scheme for R folders on your computer (Do not use Dates at the beginning of the folder name; these dates will be meaningless in a few months). Use simple, but memorable names for your folders, because you will have to type these names in your scripts.

**Section 3: Scripts** 

**Rationale:** Scripts are the files in which you write computer code and get the R software to analyze your data and generate graphics. Scripts are the ‘workhorses’ of R.

Type the following code into the script editor. Place your cursor on the line containing the code and click “Run” or use the shortcut keys **Ctrl+Enter** (Windows and Linux) or **Cmd+Return** (Mac) to run the script.

Ctrl+A to select the entire script, and then Run the entire script.

Here are some simple commands.

**Task:**

5+11

36\*3

5/9

6-24

6:24

pi

8/9+pi

pi+8/9

(pi+8)/9

Here are some other useful commands:

**+** addition

**-** subtraction

**\*** multiplication

**/** division

**^**  raise to the indicated power

Here are some other examples. To save something as an object, type in the name you want to save it as followed by the equal sign **=**.

a=5 assigns the value 5 to the object a

a returns the value for object a

b=2\*a assigns the value 2\*a to b

b returns the value for b

All of these commands generated results of simple calculations, but they did not generate anything of value that we can return to, or re-use. What we want to do is run a command and save the calculation as an entity, which R calls ‘**objects**’. **R is all about generating objects** and using those objects for the next iteration of calculations or other tasks.

**Objects**

Let’s generate an object (36\*3) and call it Object\_1. Object\_1 is 108, but if type in the following line, 108 will not appear.

Object\_1=36\*3

Only when you type

Object\_1

will the object appear as 108.

The same principle applies to long bits of code that can conveniently be saved as an object. Similarly, if we calculate the Coefficient of Variation, which is the ratio of the standard deviation relative to the mean. You can also save the Coefficient of Variation as an object. R is all about defining objects.

For example:

data= c(10,12,9,8,7,13)

mean = mean(data) # mean

sd = sd(data) # standard deviation

cv = (sd / mean)\*100

cv

**Commenting**

As you get further into data analysis and writing scripts you may think that you will remember everything you do, and why you did it, but 3-4 years later you are likely to forget. Therefore, it is useful, while the ideas are still fresh in your mind, and believe us it will save you days of agony in the future, to write yourself notes that are associated with your code. You may not need to write a comment for every line, but certainly write a comment on chunks of code, especially before the code is implemented.

For example

# Above, I transformed the response variable, and will now implement code to run a

#Generalized linear mixed model to examine the relationships between …

The hashtag symbol **#** can be used to add comments to scripts. R will not run anything to the right of **#** but will run anything to the left normally.

Here we use the hashtag to explain each line of code.

f=1/2 # f now equals one-half assigns the value 0.5 (1 divided by 2) to f

Without **#**, R will try to run the line as a single command:

g=1/8 g now equals 0.125

This will result in an error.

**Concatenate**

Objects can be lists or arrays of values. To create an object as a list, you will need to enter the list and instruct R to concatenate the list. Concatenate means linking together components in a series. The R command to concatenate is **c()**.

j=c(2,3,4) # assigns the list of numeric values 2, 3, and 4 to j

k=c(“snake”,”anole”,”gecko”,”turtle”) # assigns the character values “snake”, “anole”, #“gecko”, and “turtle” to k

Without **c()**, your input will result in an error message:

l=5,6,8

m=(“red”,”blue”,”green”)

Now try it without the close parentheses:

n=c(20,34,56,72

The command is incomplete. The > symbol on the console has been replaced with +. This indicates that R is waiting for you to complete the command. You can either complete the command, in this case with a close parenthesis or select the console and push **Esc** to abort the command. If you use **Esc**, then n is not saved so you will have to complete the script and rerun it.

**Missing data**

Any missing data values in R are indicated with NA (which means not available). NA, without quotation marks, is recognized as a missing value. “NA”, with quotation marks, is the text “NA”.

q=c(3,78,39,NA,10) # assigns the values 3, 78, 39, missing value, and 10 to q

2\*q # returns each value of q multiplied by 2, missing data cannot be

# multiplied and is still missing

**Finding help**

Help for specific commands can be found using the command **?**

?coefficients

If you don’t know the command, you can search for keywords using the command **??**

??”coefficients”

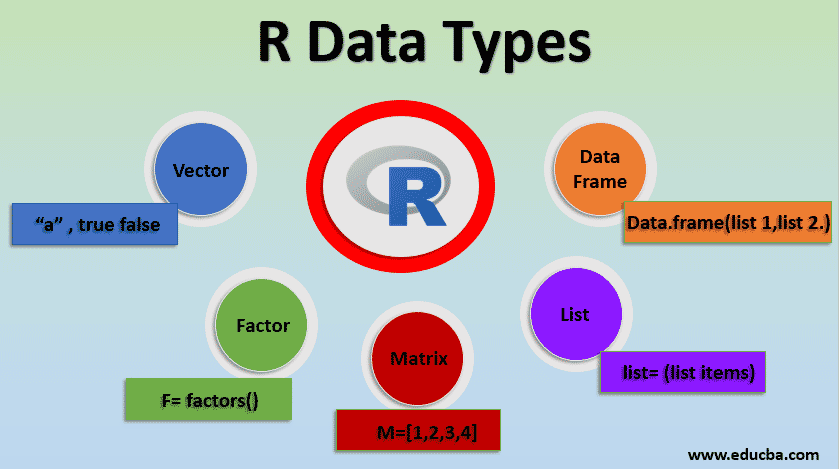
You use any search engine to find R help. R has a very helpful online community and you are likely not the first person that needs to find whatever you are looking for.

**Task:** Generate two new objects in R (using real or toy data).

**Afternoon**

**Section 4: Data and data frames**

**Rationale:** Without data, we would not be doing science. Yet, getting data is hard work, and inputting data into R can be frustrating. Data wrangling and making the data readable for R is a major skill set that should be taken seriously and with patience.



Data is often stored in databases or Excel spreadsheets and those data can be imported into the R environment. Data can also be directly input into R, but we don’t recommend that method. Data can be input and stored as one or more objects. Very often we need our data organized into tables. Data tables in R are organized such that each column is a different variable and each row is a separate observation. A small data set in R could look like this:

|  |  |
| --- | --- |
| Location | Temperature |
| one | 25 |
| two | 29 |
| three | 26 |
| four | 27 |

This data set has two variables (columns) with 4 observations (rows). Note that the top “row” shows the column names, not the data values, and is not counted among the rows of data. The data in the first column is text and the data in the second column is numeric. When different data types ( i.e., text and numbers) are mixed in the same data set, the resulting object is called a **data frame**.

The **data frame**

Most of the data you will be working within R will be imported and saved as data frames. To generate a data frame in R, use the command **data.frame()** with each column name and associated values specified in the parenthesis.

Copy, paste, and run the following script:

data1=data.frame(species=c(“carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “carolinensis”, “sagrei”, “sagrei”,“sagrei”,“sagrei”,“sagrei”,“sagrei”,“sagrei”,“sagrei”,“sagrei”,“sagrei”), length=c(16.9, 16.2, 18.2, 16.0, 14.9, 17.0, 21.1, 13.7, 15.6, 19.4,17.6, 16.9, 18.3, 20.3, 17.5, 16.7, 17.6, NA, 16.3, 18.2), weight=c(6.0, 6.1, 6.0, 5.1, 6.6, 6.2, 7.3, 4.1, 5.9, 6.9,7.5, 6.1, 5.4, 5.1, 5.4, 4.0, 5.3, 5.2, 5.5, 6.6))

You will get an error message due to the quotation marks. Text values must be in quotation marks but there are different quotation mark characters: ", “, ”, ˮ, etc. These are all considered different characters by your computer and only one will be accepted by R. The good news is using the quotation mark key on your keyboard will insert the correct symbol when you are typing in the R script editor/source pane and if you are typing in a basic text editor such as Notepad; however, if you are using any word processor that accepts text formatting such as Word, Google Docs, or Pages, then the quotation mark key will instead insert one of the other (incorrect) quotation mark symbols.

Copy, paste, and run the following script:

data1=data.frame(species=c("carolinensis", "carolinensis", "carolinensis", "carolinensis", "carolinensis", "carolinensis", "carolinensis", "carolinensis", "carolinensis", "carolinensis", "sagrei", "sagrei","sagrei","sagrei","sagrei","sagrei","sagrei","sagrei","sagrei", "sagrei"), length=c(16.9, 16.2, 18.2, 16.0, 14.9, 17.0, 21.1, 13.7, 15.6, 19.4,17.6, 16.9, 18.3, 20.3, 17.5, 16.7, 17.6, NA, 16.3, 18.2), weight=c(6.0, 6.1, 6.0, 5.1, 6.6, 6.2, 7.3, 4.1, 5.9, 6.9,7.5, 6.1, 5.4, 5.1, 5.4, 4.0, 5.3, 5.2, 5.5, 6.6))

You will not get an error message inputting the second dataset. The only difference between the two scripts is the quotation marks. You can now view the object as a data frame by typing in its name and running it as a script.

data1

For small data frames this is sufficient and relatively easy, but for larger data frames inputting manually is not recommended. Besides R will only display the first 1,000 values for an object. If you want to see details about your data frame, or another object, without displaying all of it, you can use the following commands:

**nrow(*object*)** displays the number of rows

nrow(data1)

**ncol(*object*)** displays the number of columns

ncol(data1)

**dim(*object*)** displays the number of rows and columns

dim(data1)

**names(*object*)** displays the names of columns

names(data1)

The most useful commands are ‘head’ and ‘tail’, which allow you to quickly check if your dataframe object is in the format that you intended, by observing the beginning (head) of your dataframe or the end (tail).

**head(*object*)** displays the first 6 rows unless another number is specified

head(data1)

head(data1,8)

**tail(*object*)** displays the last 6 rows unless another number is specified

tail(data1)

tail(data1,10)

**str(*object*)** displays the structure of an object including the number of rows and  
 columns as well as the first few values from each column

str(data1)

**Task:** Generate two new data frames in R (using real or toy data).

**Section 5: Variables**

**Rationale:** Variables allow us to partition the world into practical entities that we can measure and analyze. For example, we may be interested in coral bleaching, which we can measure underwater, and we could hypothesize that bleaching is related to water temperature and light intensity, which we can also measure. Coral bleaching, water temperature, and light are therefore useful quantifiable variables in a dataset.

Each column in a data frame is a variable and can be examined, manipulated, and analyzed separately from the other columns. If you know the name of the column you want to view, you can type the name of the data a dollar sign, **$**, then the name of the column. This will display the column values as a list.

***data.frame*$*column.name***

data1$length

You can also display the column by entering the column number in square brackets, **[ ]**, after typing the name of the data. This will display the entire column as a column complete with names and row numbers.

**data.frame[*number*]**

data1[2]

If you add a comma to the above **before the column number**, then it will display the column values as a list.

**data.frame[,*number*]**

data1[,2]

Note the subtle but important difference between the data when displayed as data1[3] and data1[,3]. Why can you get the mean value of the latter, but not of the former? Try it

mean(data1[3])

mean(data1[,3])

To select a range of columns by number use a colon **:**, to specify the first and last columns in the range

**data.frame[*number*:*number*]**

data1[2:3]

Likewise, rows can be extracted and viewed separately using the square brackets, **[ ]**, but you must **use a comma after** the number otherwise columns will be displayed.

**data.frame[*number*,]**

data1[12,]

You can use a colon **:**, to specify a range.

**data.frame[*number*:*number*,]**

data1[7:15,]

If you use a hyphen, **-**, R will interpret that as subtraction and either display everything or nothing.

data1[7-15,]

data1[,7-15]

If you specify a row number or range and a column number and range, you can see specific cells.

**data.frame[*number*,*number*]**

**data.frame[r*ow.range*:*row.range*,*column.range*:*column.range*]**

data1[16,3]

data1[3:8,2]

data1[9,1:2]

Data1[8:12,1:2] # result is an error, remember that R is case sensitive

**Task:** Generate two new objects in R, which will be a subset of data1.

**Section 6: Reading external data**

**Rationale:**

We have manually typed data into R but it is time-consuming, tedious, and error prone. It is far easier to enter your data in a spreadsheet, such as Excel, and read the spreadsheet into R.

The following instructions will allow you to read a file into R.

1. Make sure your data is formatted correctly
   1. Columns are variables and rows are observations
   2. All columns have unique names
   3. No spaces in column names, spaces can be replaced with periods
   4. All column names must start with a letter but can contain numbers
   5. No empty cells or missing data should be represented by NA
2. Save your spreadsheet as a .csv file (comma delimited / comma separated values)
   1. PC: select “CSV (Comma delimited)”, “CSV (MS-DOS)”, or “CSV (Macintosh)”
      1. Do not select “CSV UTF-8 (Comma delimited)”
   2. Mac: select “CSV (Comma delimited)”, “CSV (MS-DOS)”, “CSV (Windows)”, or “CSV (Macintosh)”
      1. Do not select “CSV UTF-8 (Comma delimited)”
3. If trying to import multiple worksheets from your workbook, each worksheet must be saved as a separate file
4. Use the command **read.csv()** to import the spreadsheet into R, remember to store the data as a variable otherwise, it will not be saved, the default type is data frame
   1. If the file is not in your working directory, you must specify a file path
   2. **file.choose()** will open a window so you can select a file, **read.csv(file.choose())**, this combination of commands allows easy selection of files from different folders or if you cannot remember the exact spelling of the file name
   3. *Do not use the drop-down menus in R Studio to read files into R, they will not be read appropriately*

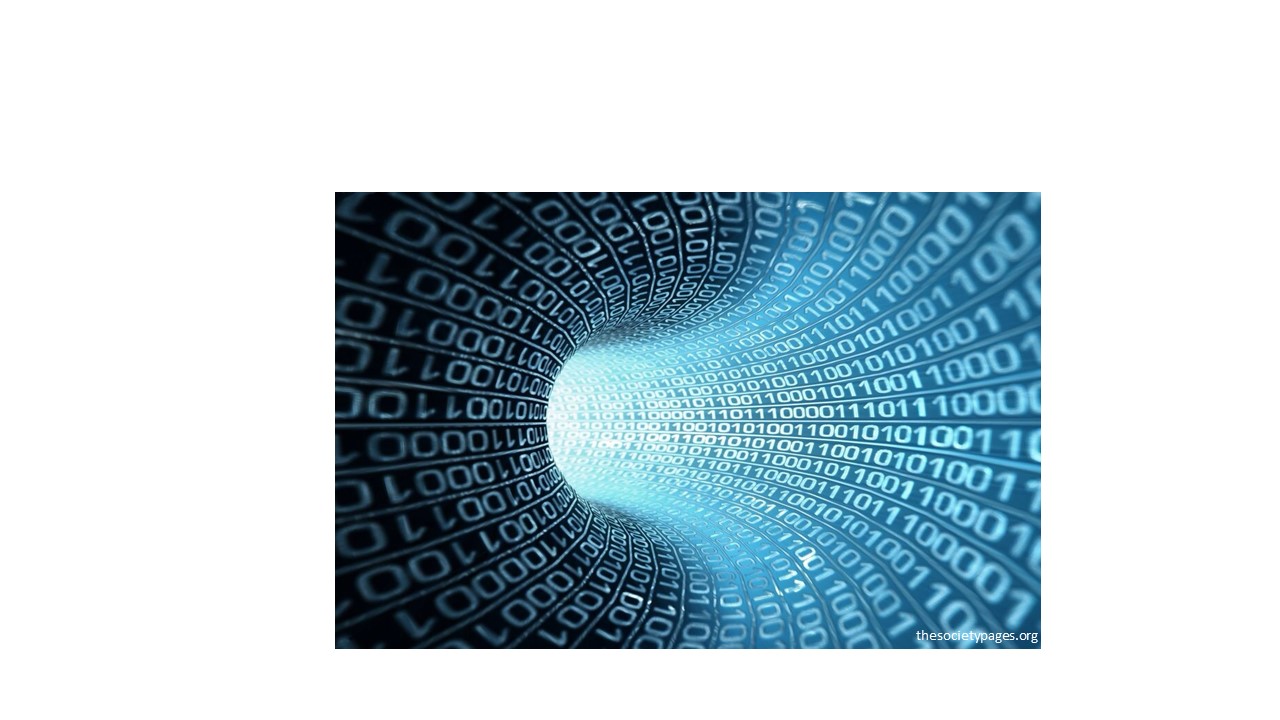
Now read the file “lakes” into R

1. Open lakes.xlsx
2. The column names contain spaces, parentheses, and slashes. These all need to be removed
   1. Change “Lake name” to “lake.name”
   2. Change “Secchi depth (m)” to “secchi.depth”
   3. Change “PO4 (ug/l)” to “po4”
3. The value for Crystal Lake PO4 is missing, enter NA into the blank cell
4. The procedure to save as a csv file varies based on the software you are using to edit the spreadsheet
   1. Excel: click on “File” then “Save As”. Under the file name you should see a dropdown menu that defaults to “Excel Workbook (\*.xlsx)”, click on this menu and select “CSV (Comma delimited)”
   2. Google Sheets: click on “File”, select “Download” then “Comma separated values (.csv)”
   3. Numbers: click on “File”, select “Export to” then “CSV …”. This will bring up the “Export Your Spreadsheet” screen, for encoding select “CSV (Comma delimited)”
   4. Calc: click on “File” then “Save As”. Under the file name you should see a dropdown menu that defaults to the current file format, click on this menu and select “Text CSV”
5. Name the imported data “lakes”
   1. Use **read.csv()**: lakes=read.csv(“lakes.csv”)
   2. Now use **read.csv(file.choose())**: lakes=read.csv(file.choose())
      1. You will not be able to do anything else in R or RStudio until you close the “Select file” window
      2. The “Select file” window may not pop up. It may open minimized or behind another window. You will likely need to look for it.
      3. Once found, use the “Select file” window to navigate to the lakes.csv and open the file
      4. If you opened the “Select file” window by mistake, you can close it without opening a file by clicking “Cancel”
6. View lakes to ensure it is read correctly

Step 5, above, illustrates two methods to read files into R. You only need to use one of them. You do not need to do both. The imported data will be a data frame.

*Do not use the “Import Dataset” menu in RStudio. It will not read your data correctly.*

**Section 7: Producing Outputs**

**Rationale:** Producing objects, such as a dataframe, and graphical outputs is the major task of data analysis. You can (and should) save objects, especially data frames, because if the computer gets turned off all information will be erased from your Console.

Some days you may have worked hard to write code to generate a new dataframe or another new object. It is often a good idea to save the object, or even save the dataframe as a new csv file. When you return to your coding task, then you do not have to repeat all the analytical steps that you did the previous day. You simply ‘call up’ the dataframe or the object, and continue with your analysis.

You can save a dataframe or other object simply by using

save(myObject, file = “My\_file\_name”)

These commands will save the object (myObject) in the current working directory, as ‘My\_file\_name’. **The following day, go to the working directory,** and simply type in

load(My\_file\_name)

And the object will return to your environment.

You can also save a dataframe as a csv file using the following:

write.csv(Name\_of\_DataFrame, "Path to export/My\_File\_Name.csv").

To read the file the next day, type:

YesterdaysObject = read.csv("My\_File\_Name.csv", header=TRUE)

**Incidentally, you can use = and <- interchangably (the latter is a remnant of the R language before the year 2021).**

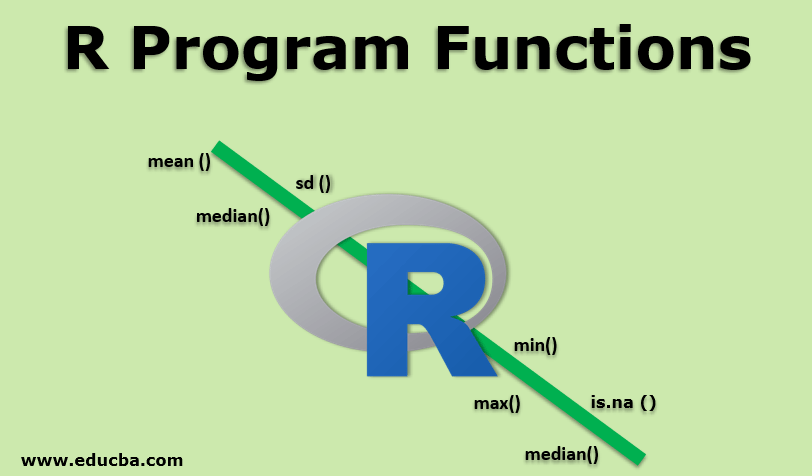
**Task:** Save the most important objects that you generated today.

**Day 2**

**Morning**

**Section 8: Functions**

**Rationale:** Functions are self-contained pieces of script that contain input statements (arguments) to systematically compute a result and generate a new object. Functions can be as simple as averaging across a set of numbers or as complicated as pre-testing a rocket before a space flight.



A function is similar to an equation but instead of reading from left to right, a function is read from right to left to form a new object, using the name on the left.

For example,

alpha=0.5

beta= 0.7

x=c(1,2,3,4,5,6,7)

New\_Object\_1 = alpha + beta\*x

The input command on the right gets executed and a new object is generated. Another example is to use an in-built function, for example, ‘*mean’.*

values=c(24,35,56,82)

mean(values)

The ‘mean’ function: 1) adds the values, 2) counts the number of values, then 3) divides the sum of the values by the number of values. **Many of the actions that we will be requiring R to perform will be via functions.**

Below are some commonly used data summary functions:

**sum(*values*)** adds the indicated values

sum(data1$weight)

**prod(*values*)** multiples the indicated values

prod(data1$weight)

**min(*values*)** finds the smallest value from the indicated values

min(data1$weight)

**max(*values*)**  finds the largest value from the indicated values

max(data1$weight)

**range(*values*)** finds the smallest and largest values from the indicated values

range(data1$weight)

**mean(*values*)** calculates the mean of the indicated values

mean(data1$weight)

**median(*values*)** calculates the median (middle value) of the indicated values

median(data1$weight)

**sd(*values*)** calculates the standard deviation of the indicated values

sd(data1$weight)

**mad(*values*)** calculates the median absolute deviation of the indicated values

mad(data1$weight)

Most functions will return “NA” (not available or not applicable) or “NULL” (no value) as an output if the input values include missing data. Many functions will accept the argument **na.rm** which removes (rm) missing values (na) when used/activated (=TRUE). The full argument is typed in your script as **na.rm=TRUE** after the values and separated from the values with a comma.

sum(data1$length) output is NA

sum(data1$length,na.rm=TRUE) output is the summation of the non-missing values

median(data1$length) output is NULL

median(data1$length,na.rm=TRUE) output is the median of the non-missing values

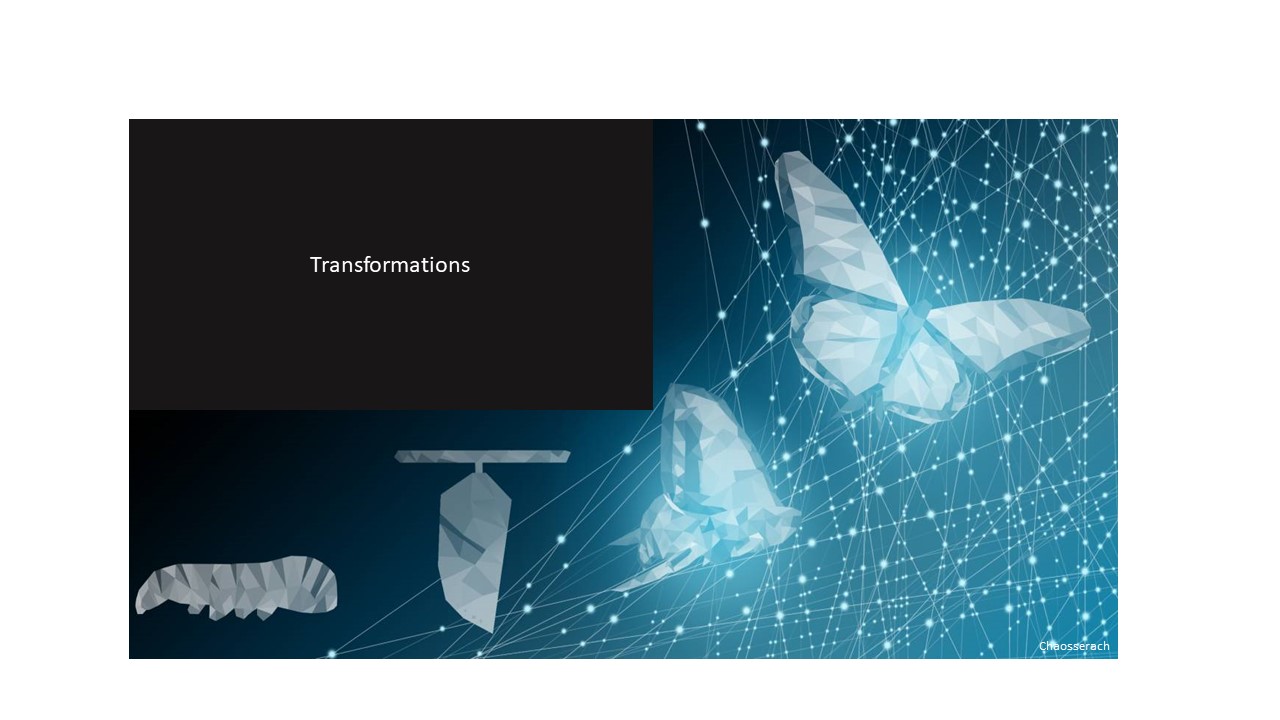
**Task:** Why can’t we get the mean value of data1[,2]? How would you fix the code to calculate the mean of variable 2 as the data1 object?

The outputs of functions can be saved as an object by typing the object name followed by **=** then the complete function.

**object.name=function(*values*)**

weight.mean=mean(data1$weight)

length.range=range(data1$length,na.rm=TRUE)

**Section 9: Transformations**

**Rationale:** Some raw datasets need to be transformed to fit the distributions of the underlying statistical models.

R contains a variety of mathematical transformations and constants. Knowing them can make your life much easier.

**sqrt(*value*)** square root

**log10(*value*)** common logarithm (log base 10)

**log2(*value*)** binary logarithm (log base 2)

**log(*value*)**  natural logarithm

**log(*value*,*base*)** logarithm to the indicated base

**pi** Pi rounded to six digits = 3.141593

**exp(*value*)** e (rounded to six digits = 2.71828) raised to the indicated value

The square root (second root) is the only root with a function. If you wish to find a different root, you can use fractional exponents.

***number*^(1/3)** cube root (third root)

27^(1/3)

27^1/3

***number*^(1/4)**  fourth root

1296^(1/4)

These functions can be used on lists and columns within data frames. We have previously applied a function to a list, now let’s do the same for a column in data1.

log10(data1$weight) the result is a list of the common logarithms of the weights

sqrt(data1$length) the result is a list of the square roots of the lengths

Note: the value of NA is still NA

exp(data1$species) the result is an error message,

Note that **mathematical functions cannot be applied to text**

We can combine mathematical functions and save them as objects.

weight.log=log(data1$weight+1) This will save the natural logarithm of the weights +1,

i.e. 1 is added to the weights then the log is taken

**Task:** Generate a series of 10 numbers, call it an object and natural log transform all 10 numbers.

**Section 10: Combining columns and rows**

**Rationale:** Sometimes we need to combine datasets for data analysis; combining data can involve new data of the same type, by adding rows, or new variables, and by adding columns.

We often need to add rows or columns to existing data frames or combine lists to make new data frames. We can combine objects as rows or as columns: **rbind(*list of objects*)** will combine as rows and **cbind(*list of objects*)** will combine as columns. Adding new rows requires that the number of columns in both datasets be the same, and adding columns requires that the number of rows be the same. Note that you can add data as rows or columns if they are multiples of other objects, but many problems can arise (see below).

row1=2:7 row1 as a list of the values 2 through 7

row2=10:15 row2 as a list of the values 10 through 15

rows1.2 =rbind(row1,row2) rows1.2 is a table that has as rows row1 and row2

Now try it with objects that have a different number of values.

row3=31:38

rows1.3=rbind(row1,row3) result is an error message

Try again with objects that have a different number of values but the number of values in the larger rows is a multiple of the number of values in the smaller rows.

rows1.2.4=rbind(row1,row2,row4) result is an error message because we have not created row4

row4=31:33

rows1.2.4=rbind(row1,row2,row4) the new object is a table of row1, row2 and row4 as rows

but the values of row 4 are repeated; therefore, be careful and determine if this repeated data is what you want.

In each successful instance of rbind, the result is a data frame with the indicated rows in the listed order. The rows are named as the original object and the columns are numbered. We can display, change, or remove the column names using **colnames(*object*)**, as follows:

colnames(rows1.2.4) result is NULL because there are no column names

colnames(rows1.2.4)=c("gala","honeycrisp","mcintosh","fuji","jonagold","granny smith")

this adds column names to rows1.2.4

Rownames can be displayed, changed, or removed using **rownames(*object*)**.

rownames(rows1.2.4) displays the row names

rownames(rows1.2.4) =NULL removes the row names

The same format is used for cbind.

column1.2=cbind(row1,row2) column1.2 is a table that has as columns row1 and row2

column2.1=cbind(row2,row1) column2.1 is a table that has as columns row2 and row1

column1.4.2=cbind(row1,row4,row2) column1.4.2 is a table with columns row1, row4

(with values repeated) and row2

The same format is used to add rows or columns to data frames.

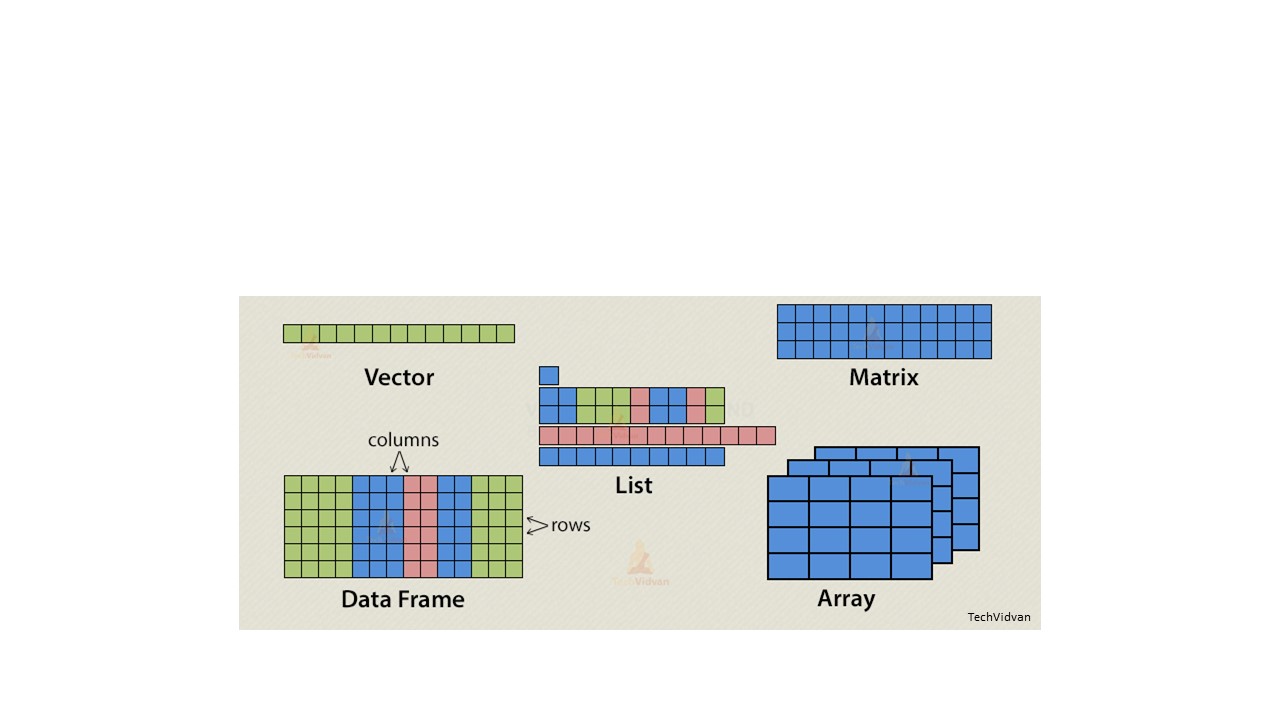
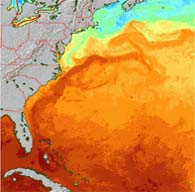
data1a=cbind(data1,weight.log) adds weight.log to data1, the column name for weight.log is weight.log

**Task:** Generate 2 data frames and combine them as a new object.

**Afternoon**

**Section 11: Vectors and matrices**

**Rationale:** Data come in many forms and the most common forms are vectors (list of data elements of the same type) and matrices (list of data elements of the same type arranged as a fixed number of rows (*i*) and columns (*j*)). For example, an image from a camera, phone, or satellite can be displayed as a matrix, because every image pixel has a value.

An R vector is a list of items in a specific sequence. We have created several vectors previously.

j=c(2,3,4)

k=c(“snake”,”anole”,”gecko”,”turtle”)

q=c(3,78,39,NA,10)

j & q are vectors containing numbers

k is a vector containing text

In each instance, all of the items are the same type of data, i.e. numbers (i.e., j and q) or text (i.e., k).

A new, empty vector can be created using **vector(length=*desired length*)**. Empty vectors can later be filled in with values, generated from functions or other methods.

vector1=vector(length=10) this creates an empty vector with 10 items

You can create a regular sequence of values using

**seq(*starting value*, *maximum value*,by=*interval*)**.

sqnc1=seq(1,9,by=2) creates sqnc1 as the sequence starting at 1 with a maximum value of 9 increasing by 2

sqnc2=seq(0,13,by=pi) creates sqnc2 as the sequence starting at 0 with a maximum

value of 13 increasing by pi, note that the sequence ends at

12.566371 not at 13, the maximum value may or not be part of the final sequence

If you omit the argument **by**, the interval defaults to 1.

seq(0.5,11.5)

Each of these sequences is a list which makes them vectors

You can create replicated values using **rep(*values*,each=*number*,length.out=*number*,times=*number*)**

The arguments above (i.e., each, length.out, and times) do not have to be given each time the command is used but each conveys important instructions to the command: **each** is the number of times to repeat each value before proceeding to the next value, **length.out** is the number of values to generate, **times** is the number of complete replications to generate. If you omit **each** and/or **times**, they default to 1. If **length.out** is omitted, the replication completes to its maximum size based on the other arguments. If both **length.out** and **times** are assigned values, then **length.out** takes precedence.

Some more examples of vectors include:

rplc1=rep(2:7,each=2) creates rplc1 as the repeated values 2 through 7

each value repeated 2 times

rplc2=rep(2:8,each=2,length.out=10) creates rplc2 as the first 10 items of the values

2 through 8 each repeated 2 times

rplc3=rep(4:12,each=3,times=5) creates rplc3 as the repeated values 4 through 12

each value repeated 3 times but that entire

sequence is repeated 5 times

rep(11:20,times=5) the sequence 11 to 20 repeated 5 times

rep(11:20,length.out=21,times=5) the sequence 11 to 20 repeated 5 times but only until 21 items are generated

Each of these sequences is a list which makes them vectors

An R matrix is a rectangular arrangement of items. Like a vector, all of the items must be the same type of data. Matrices have rows and columns. We have created matrices previously when we used rbind and cbind: rows1.2.4, column1.4.2, and all other objects created in section 9 are matrices.

A **matrix** can also be created using

**matrix(*values*,nrow=*number of rows*,ncol=*number of columns*)**

The number of values must equal the number of cells in the matrix or an error will result. The values will be filled in by column. The argument **byrow** can be set to TRUE if you want the values to be filled in by row. Alternatively, only 1 value can be given and it will populate all the cells in the matrix.

mtx1=matrix(1:12,nrow=4,ncol=3) creates a 4x3 matrix with the values 1-12

mtx2=matrix(1,nrow=4,ncol=2) creates a 4x2 matrix with 1 in all the cells

mtx3=matrix(c(2,4,7,8,11,13,7,8,2,4),nrow=2,ncol=5)

creates a 2x5 matrix with the specified values

mtx4=matrix(c(2,4,7,8,11,13,7,8,2,4),nrow=2,ncol=5,byrow=TRUE)

creates a 2x5 matrix with the specified values with the values filled in by row

matrix(1:10,nrow=5,ncol=5) error: not enough values

matrix(1:10,nrow=2,ncol=3) error: too many values

The commands **seq** and **rep** can be used in matrices.

matrix(seq(2,32,by=2),nrow=4,ncol=4) 4x4 matrix with the sequence 2 to 32 increasing by the interval 2

matrix(rep(51:55),each=4,nrow=4,ncol=5) error: first close parenthesis is in the wrong place

matrix(rep(51:55,each=4),nrow=4,ncol=5) 4x5 matrix with the values 51 to 55 each repeated 4 times

An empty matrix can be created by omitting the values. Empty matrices can later be filled in with values generated from functions.

mtx10=matrix(nrow=5,ncol=12) this creates an empty matrix with 5 rows and 12 columns

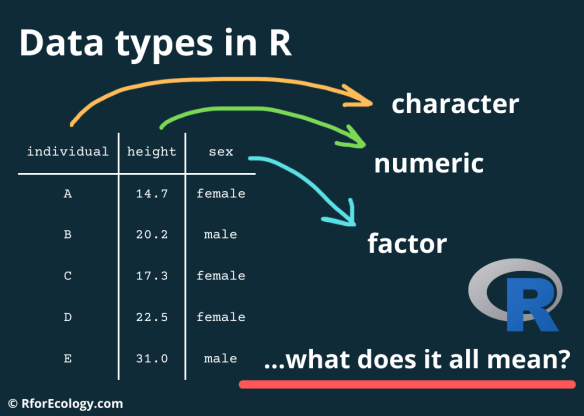
An R data frame is a rectangular array of items but unlike a matrix, the items can be of different types. A data frame has rows and columns. The items in the same column must be the same type of data but different columns can contain different types of data. data1 is a data frame. The file you will read into R in the next section is a data frame.

**Task:** Generate a 3 by 4 matrix as a new object.

**Section 12: Variable types**

**Rationale:**

Datasets imported into R may contain variables that are inappropriately classified. We need to check the variables (i.e., columns) of our data matrices to determine that we are about to analyze the appropriately classified variables.



Many functions in R only work when given the correct type of data. This not only includes how the data is saved (i.e., vector, matrix, or data frame), but also the classification of the data, for example, whether the data is numeric, character, or logical.

Numeric data are numbers, whereas integers are whole numbers, without decimals. The variable a, the vector j, and the columns' length and weight in data1 are all numeric.

Character data is text. The variable c, the vector k, and the column species in data1 are all characters.

Logical data is TRUE or FALSE. Not the words “true” and “false” but binary data indicating some statement is TRUE or FALSE. R can treat logical data as numeric data with TRUE assigned the value 1 and FALSE assigned the value 0. To see logical values, run 5<8 and 9=23. The second command, 9=23, will result in an error. R is trying to set the variable 9 equal to 23 but variable names cannot begin with numbers. For a logical test use double equal signs **==**, run 9==23.

To see the variable type use the command **class(*data*)**.

class(a)

class(k)

class(data1$length)

class(5<8)

Variables can be converted from one type to another, with limitations.

**as.character(*variable*)** will convert numeric and logical data to character data

**as.numeric(*variable*)** will convert logical data and character data using number

characters to numeric data, character data that uses letters cannot be converted to numeric

**as.logical(*variable*)** will convert 0, 1, “false”, and “true” to logical data

Creating variables

numb1=c(3,4,23)

numb2=c(0,0,1,1,1)

char1=c("FALSE","TRUE")

char2=c("false","true")

logic1=c(TRUE,TRUE,FALSE)

Checking the class of the variables

class(numb1)

class(numb2)

class(char1)

class(char2)

class(logic1)

Changing the class of the variables

numb1=as.character(numb1)

numb2=as.logical(numb2)

char1=as.logical(char1)

char2=as.logical(char2)

logic1=as.numeric(logic1)

Viewing the variables

numb1 the numbers are surrounded by quotation marks

numb2 the data now displays as FALSE, FALSE, TRUE, TRUE

char1 the data now lacks quotation marks

char2 the data now lacks quotation marks, and the words true and false are uppercase

logic1 the data now displays as 1, 1, 0

Checking the new class of the variables

class(numb1)

class(numb2)

class(char1)

class(char2)

class(logic1)

Statistical analysis adds an additional data type: **factor**. Factor data can be numbers, characters, or logical values but there must be a limited number of unique values. Factor data are treated as categories and this distinction is only important for some analysis methods such as Analysis of Variance (ANOVA). For example, you may have data on benthic composition near a hotel development, and at some distance from the hotel development, before and after the development. You may be interested to see whether the hotel developed influenced the benthic composition. You should use **factor** to categorize your data (later in your degree you will learn about fixed and random effects, which are more advanced concepts but crucial to know).

These methods usually treat character and logical data as factors by default but will treat numeric data as numbers. The argument **factor(*data*)** can be inserted into the functions for these methods or you can use **as.factor(*data*)** to convert the data into a factor.

category1=c("red","blue","green")

category1

class(category1)

category1=as.factor(category1)

category1 displaying factors will show the data as well as a list of all unique values for that data after “Levels”

category2=c(44,66,88,120,44,66,88,120,44,66,88,120)

class(category2)

category2=as.factor(category2)

class(category2)

category2 even though each value is repeated, the “Levels” only lists each value once, this is because “Levels” is a list of unique values

Matrices can be converted to data frames using **as.data.frame(*object name*)**.

rows124=as.data.frame(rows1.2.4) creates rows124 as a data frame with the data from rows1.2.4

rows124 and rows1.2.4 contain identical data but rows124 is a data frame and rows1.2.4 is a matrix

rows1.2.4=as.data.frame(rows1.2.4) converts rows1.2.4 from a matrix to a data frame

rows124 and rows1.2.4 are identical, the same data, and now both are data frames

Data read into R using **read.csv** will be a data frame. You can convert it to a matrix using **as.matrix(*object name*)** or read it as a matrix using **as.matrix(read.csv("*file.csv*"))**. For example, we will read the file “late spring.csv”.

notamatrix=read.csv("late spring.csv") read “late spring.csv” into R

class(notamatrix) notamtrix is indeed not a matrix

amatrix1=as.matrix(notamatrix) create amatrix1 using the data from notamatrix

class(amatrix1) amatrix1 is a matrix

amatrix2=as.matrix(read.csv("late spring.csv")) read “late spring.csv” into R as a matrix

class(amatrix2) amatrix2 is a matrix

Here is a simple example of setting up a dataframe with one variable as a factor.

*A=c(1,1,1,2,2,2)*

*B=c(1,2,3,4,5,6)*

*D=cbind(A,B)*

*D1=data.frame(D)*

*D1$A=as.factor(D1$A)*

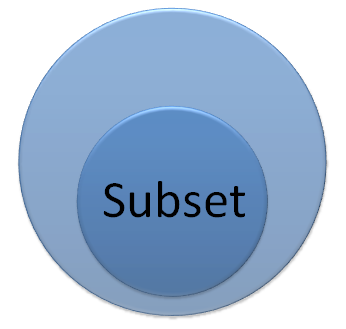
*D1*

*str(D1)*

Generate a 16-row 2-column data frame in R. One column must be type factor and the other type numeric.

**Day 3**

**Morning**



**Section 13: Subsetting**

**Rationale:**

We may wish to only analyze a part of a large dataset. Therefore, we need to know how to take subsets of a dataset.

So far the data we have entered or read have worked, however, we may wish to only analyze a part of the data. R has many built-in data sets that can be used as you learn and become comfortable with subsetting data and objects. Subsetting is extracting a portion of a larger data set. This usually occurs when you need to work with observations that have a specific value or range of values for one or more variables. For the rest of this section, we will work with the data set **CO2**. Information about this data set can be found in R help.

CO2 displays the data set CO2

?CO2 displays information about CO2

data() lists the data sets in the package “datasets”

Basic subsetting can be done using square brackets **[ ]** to select specific rows, here 1 to 18.

co2.rows1.18=CO2[1:18,]

You can select rows that correspond to a specific value from one of the columns ***new.object=object[object*$*variable==value,]***

co2.conc1000=CO2[CO2$conc==1000,] #note the double equal sign refers to selecting

values of that conc that are 1000.

Inequalities can also be used

co2.conc.low=CO2[CO2$conc<500,]

Square brackets **[ ]** only work if you are specifying a single condition for a single column. If you need to specify multiple conditions and/or multiple columns then use **subset(object,condition)**.

co2.conc1000a=subset(CO2,conc==1000) same as co2.conc1000

co2.conc.lowa=subset(CO2,conc<500) same as co2.conc.low

If specifying multiple conditions that both must be met (and) join them using an ampersand **&**

co2.nonchilled.highconc=subset(CO2,conc>500&Treatment=="nonchilled")

co2.conclow.mis.uptkhgh=subset(CO2,conc<650&Type=="Mississippi"&uptake>20)

If specifying multiple conditions where either can be met (or) join them using a vertical bar **|**

co2.conc.midlow=subset(CO2,conc==500|conc==95)

co2.concmid.quebec=subset(CO2,conc==500|Type=="Quebec")

If reversing a condition, i.e., not equal to or not greater than, insert an exclamation point, **!**, before the condition

co2.conc.not1000=subset(CO2,!conc==1000) #conc not equal to 1,000

co2.conc.mid=subset(CO2,conc<700&!conc<400) #conc less than 700 and not less than 400

If only selecting certain columns, use the argument **select**

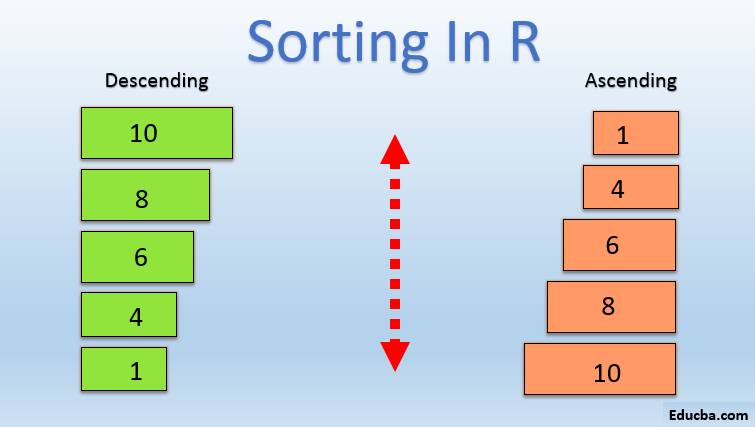
co2.qub.non.conchigh=subset(CO2,conc>500&Treatment=="nonchilled"&Type=="Quebec",

select=c(Plant,conc,uptake))

conc greater than 500 and nonchilled and Quebec only columns Plant, conc, and uptake

**Section 14: Sorting**

**Rationale:**

Occasionally we may wish to change the order of a variable, specifically to generate a table or a figure that seems more orderly trending from small to large numbers or vice versa. Care must be taken, however, so that all other variables are changed to the new order. 

To order or sort a vector the command, **sort** can be used. The **sort** command can order a vector ascending or descending and can place missing data (NA) at the beginning, end, or omit it.

sort(q) # the values in the vector q ordered from

smallest to largest and NAs are omitted

To reverse the order, from largest to smallest, the argument **decreasing** is set to TRUE (**decreasing=T**).

sort(q,decreasing=T) # the values in the vector q are in order from largest to smallest and NAs are omitted

To include NAs, the argument **na.last** can be set to TRUE or FALSE (**na.last=T** or **na.last=F**). Setting **na.last** to TRUE places the NAs last and setting **na.last** to FALSE places the NAs first. The default is **na.last=NA**, which omits the NAs.

sort(q,decreasing=T,na.last=T) # the values in the vector q are order from largest to

smallest and NAs are placed at the end of the list

The major problem with **sort** is that it will only work with a single vector/list/variable. If you want to sort a data frame by the values of one or more of the columns, then you will need to use **order**. The syntax to sort a data frame is ***data.frame.name*[order(*column(s).to.sort.by*),]**, which is the name of the data frame you are sorting, open square bracket, **order**, open parenthesis, the column(s) you want to sort by in order of priority separated by commas, close parenthesis, comma, close square bracket.

CO2[order(CO2$conc),] # this sorts CO2 by the column conc

Tied values are preserved in their original order. If you want to break ties by the values of another column, you enter the secondary column after the primary column.

CO2[order(CO2$conc,CO2$uptake),] # this sorts CO2 by the column conc, ties are

sorted using uptake, further ties are preserved in their original order

You can add as many columns as you like to continue breaking ties.

In each of the above order examples, the order is ascending, smallest to largest. To reverse this, sort largest to smallest, and place a negative sign in front of the column name.

CO2[order(-CO2$conc,CO2$uptake),] # this sorts CO2 by the column conc but in

descending order, ties are sorted using uptake in ascending order, and further ties are preserved in their original order

If you are sorting your data, you likely want to save the result.

co2.sortconc=CO2[order(CO2$conc,-CO2$uptake),]

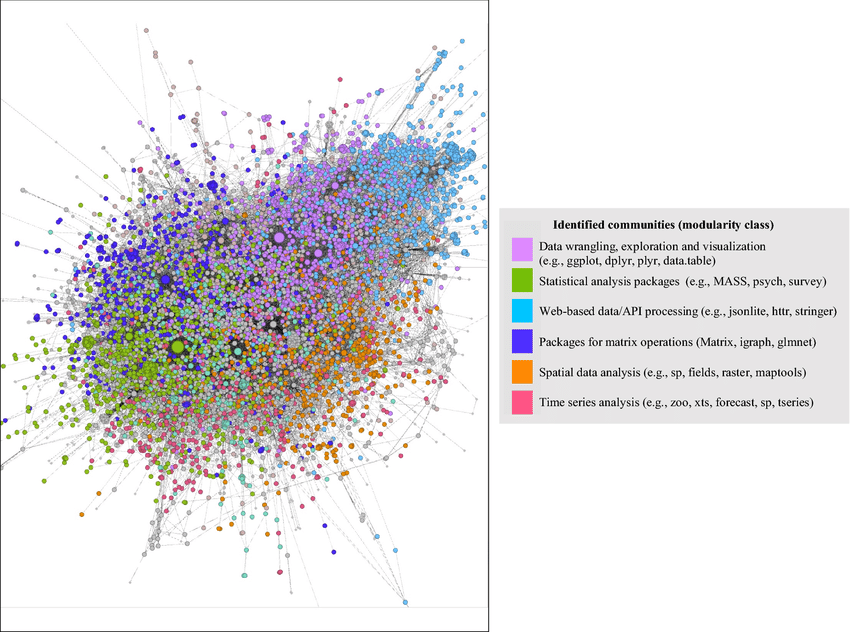
# the object co2.sortconc is CO2 but sorted by conc ascending then uptake descending

**Task**: Using **order**, sort one of the data frames you created in Section 9 by a column of your choice.

**Afternoon**

**Section 26: Packages and libraries**

**Rationale:**

Some of the world’s best statisticians have written packages that contain useful R code to run analyses. There are over 20,000 R packages. The great thing about R packages is that they are written by statisticians but the worst thing about R packages is that they are written by statisticians. 

**plyr**

plyr is a package that is very useful for deriving summaries of some variables in a large dataset. The dataset is called islands\_data, and we are interested in the predictive variables Island and Habitat, using the observed data called LLC (i.e., Live Coral Cover). This is how we can quickly generate summaries for data with thousands of rows.

install.packages(“plyr”) # code to install the plyr package

library(plyr)

#Read in the csv file called islands\_data

sumdata = ddply(islands\_data, c("Island","Habitat"), summarise,

N = length(LCC),

sum = sum(LCC),

mean = mean(LCC),

sd = sd(LCC))

sumdata

**psych**

Another way to get simple summaries of a variable is to use the package psych.

Install the package “psych”

library(psych)

describe(islands\_data$LCC)

If you only want to display the mean, minimum, maximum, range, and standard deviation of a variable.

describe(islands\_data$LCC, fast=TRUE)

**Section 16: Summary tables**

**Rationale:**

Often we need to make summary statistics of our variables, to either check the distribution of a variable or to make presentable tables.

**summary**

To quickly observe the summary of a variable, you can use the function ‘summary’.

You will get the minimum, median, mean, and maximum, first quartile, and 3rd quartile of the variable. For example, let’s make a new object Bvd, and view a summary of the object

Bvd=c(2,3,4,7,10)

summary(Bvd)

The function ‘summary’ can also be used to observe the outputs of a model, for example, a Linear or a Generalized Linear Model. Although we won’t be running such models in this short course, make sure to keep ‘summary’ as a function to quickly observe modeling results later in your graduate classes.

**table**

The function ‘table’ is a useful way to generate **frequency tables**. If you have some variables and you are interested in counting how many data points are in each level of a variable, then **table** is a go-to function. For example, the toy data below outlines 10 study sites and records whether the corals bleached (Y) or did not bleach (N) this year in August. You may wish to know how many sites bleached and how many sites did not bleach.

Bleaching=data.frame(Site= c(1,2,3,4,5,6,7,8,9,10), Bleach.August=c("Y","N","N","N","Y","Y","Y","Y","Y","Y"))

table(Bleaching$Bleach.August)

**Section 17: Data manipulation**

**Rationale:**

Some datasets are so large that you may need some simple summaries, such as means or standard deviations. Or you may want to group some adjacent datasets and analyze the mean. There are lots of reasons why you may want to manipulate data.

Often you need to see descriptive values for several columns in your dataset at the same time. You can write separate scripts for each variable or you can use **sapply**. This will apply the same command to all of the indicated columns.

sapply(CO2[,4:5],FUN=mean) # applies **mean** to CO2 columns 4 through 5

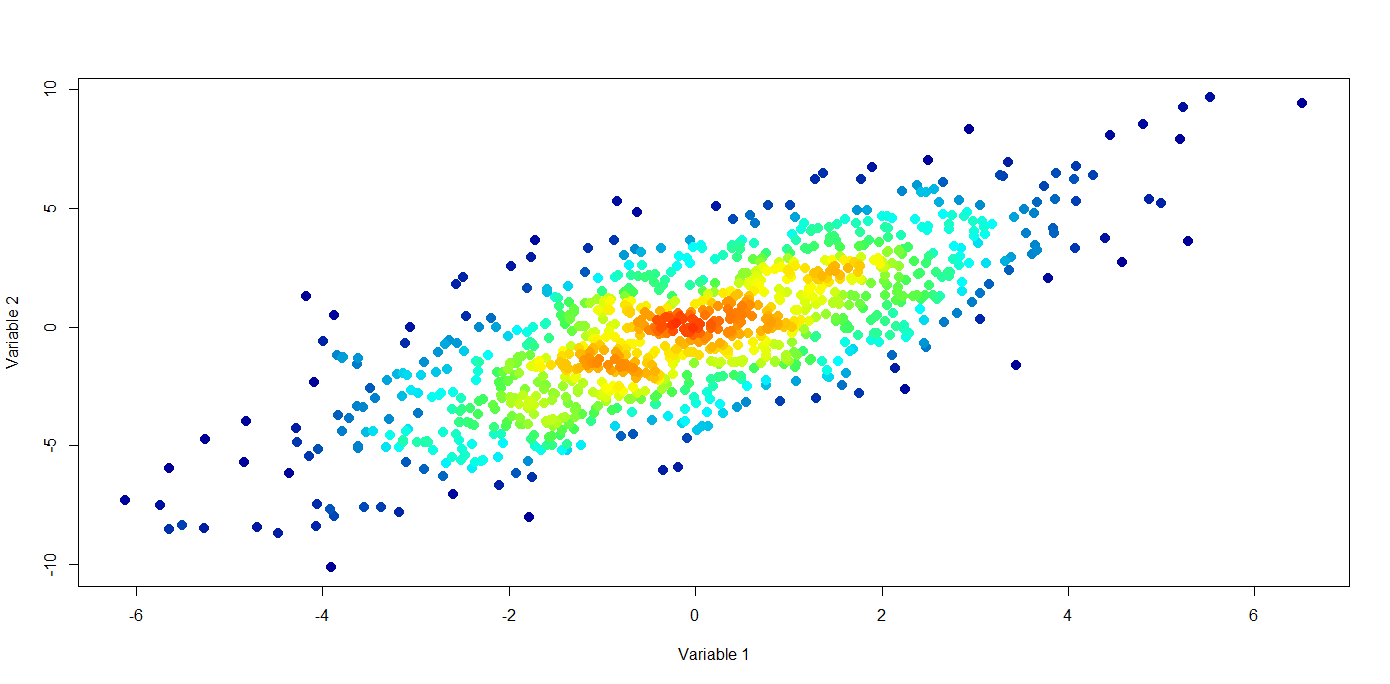
This is adequate if you want to apply the function to the entire column; however, some values of that column may belong to one factor and other values to a separate factor. To apply a function to one column (a response) based on values from a separate column (a factor) you can use **tapply**.

tapply(CO2$uptake,CO2$Treatment,FUN=mean) # applies **mean** to uptake after subdividing uptake by Treatment

**Day 4 (plotting)**

Morning

**Section 18: Scatterplots**

**Rationale:** 

Visualizing data can unveil tremendous insight. We should always start exploring data with a simple scatterplot.

**Scatterplots** are graphs showing the (x,y) coordinates of points. There are several graph generating functions in R and one of the most flexible is **plot**. **plot** will generate an appropriate graph based on the type of data used. If **plot** is given two numeric variables of the same length, it will generate a scatterplot.

Use the data **faithful** which is the waiting time between eruptions and duration of eruptions for the Old Faithful geyser in Yellowstone National Park.

plot(faithful$waiting,faithful$eruptions)

The above code generates a scatterplot showing waiting times and eruption durations. This is a very rough scatterplot, but we can improve it. The first thing we can do is add labels to the x and y-axes using the arguments **xlab** and **ylab**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)")

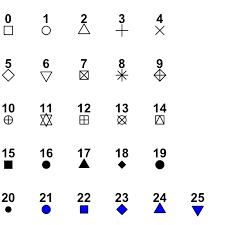
This is the same plot but now the axes have labels. Note that everything placed in quotation marks will be part of the label. Next, we can adjust the axis ranges. By default, R will use the minimum and maximum values as the axis range. The range, again by default, is extended in each direction by 4%. To specify a desired range use the arguments **xlim** and **ylim**. You can also omit the extra 4% by setting **xaxs** and **yaxs** equal to i: **xaxs="i"** and **yaxs="i"**. The minimum and maximum values for faithful$waiting are, respectively, 43 and 96, so 40 and 100 would make a good range for our x-axis. The minimum and maximum values for faithful$eruptions are, respectively, 1.6 and 5.1, so 0 and 6 would make a good range for your y-axis.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6))

Our axes still have the extra 4% because we did not use the arguments **xaxs="i"** and **yaxs="i"**. These two arguments only have two possible values r, regular, and i, interval, the default value is r which adds the extra 4%.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i")

Now let's change the symbols used for the points. The argument is **pch** and it is set to a numeric value from 0 to 25. To see a key for how symbols and numbers correspond use the help common: **?pch**. For solid circles, we will use **pch=16**



plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)", xlim=c(40,100), ylim=c(0,6), xaxs="i", yaxs="i", pch=16)

If you want your points to be a different color you can use the argument **col** and type in the name of the color you want. The command **colors()** will return a list of color names.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)", xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

Data for the plot can be entered in two formats. The format we have used is x-axis variable comma (,) y-axis variable. Alternatively, you can use the format: response variable (y-axis) tilda (~) predictor variable (x-axis). This format is the same as used for modeling functions but generates plots identical to the first format we used.

plot(faithful$eruptions~faithful$waiting,xlab="Waiting time (min)",ylab="Eruption duration (min)", xlim=c(40,100), ylim=c(0,6), xaxs="i", yaxs="i", pch=16, col="red")

These last two plots use different data formats but generate identical plots.

**Section 19: Plotlines**

**Rationale:**

We may wish to lines instead of data points.

Many scatterplots have lines added to them. These lines can be fitted lines from model output, reference lines, or lines to divide the plot into sections. To add one or more straight lines to an existing scatterplot, the commands **segments** and **ablines** can be used.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

This will remake the final scatterplot from the previous section.

If you know the (x,y) coordinates of the line to be added you can use **segments** and enter the coordinates of one end of the line and then the other end of the line: **segments(*x1*,*y1*,*x2*,*y2*)**. The minimum values of waiting and eruptions are 43 and 1.6 respectively, and the maximum values of waiting and eruptions are 96 and 5.1 respectively. To add a line from the minimum values to the maximum values:

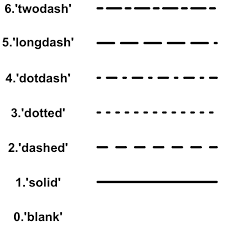
segments(43,1.6,96,5.1)

Now let's improve the line. First, generate a new plot

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

Now make the line wider using **lwd**. Please note the interpretation of **lwd** is device-specific so you may need to experiment with several different values.

segments(43,1.6,96,5.1,lwd=3)

Line type can be changed using **lty**. The values for **lty** can be solid (the default), dashed, dotted, dotdash, longdash, twodash, or blank. Blank will draw an invisible line

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",

xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

segments(43,1.6,96,5.1,lwd=3,lty="dotdash")

Once again we can use **col** to change the color.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

segments(43,1.6,96,5.1,lwd=3,lty="dotdash",col=”blue2”)

A second command to add lines to a plot is **abline**. With **abline** you can specify a slope and intercept, a x-coordinate for a vertical line, or a y-coordinate for a horizontal line. Like **segments**, **abline** adds a line to an existing plot and accepts the arguments **lwd**, **lty**, and **col**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

abline(h=3.5) adds a horizontal line (h) through the y-coordinate 3.5

abline(v=67) adds a vertical line (v) through the x-coordinate 67

Straight lines have both a slope (steepness of the line or the number of y-units that increase as the x-units increase by 1) and an intercept (the y-coordinate where the line intersects the y-axis). These values can be used to specify a straight line using **abline** where a = the intercept and b = the slope. The line we made using segments (above) has an intercept of -1.25 and a slope of 0.066

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

abline(a=-1.25,b=0.066) the line extends across our entire plot

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",

xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

abline(a=-1.25,b=0.066,lwd=2,lty="dashed",col="steelblue")

Same as previous but the line is formatted.

Thus far we have added straight lines to an existing scatterplot. We can add any line or curve that can be expressed as a mathematical function using **curve**. **curve** is also a standalone function that can generate a plot; as a result, if you want to use **curve** to add a line or curve to an existing plot you must use the argument **add=T**, add equals true.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

curve(-1.25+0.066\*x,add=T)

This is the same line we added using **abline**.

Straight lines don’t fit this data especially well so a curve may be more appropriate. Plotting a curve requires entering the appropriate function into **curve**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,6),xaxs="i",yaxs="i",pch=16,col="red")

curve(0.4\*1.03^x,add=T)

Now we have a curve.

The default beginning and end points correspond to the min and max for our x-values +4%. Our current plot cannot fit the curve as the final point on our curve is 7.24 but our y-axis is from 0 to 6. We can extend our original plot and format our curve.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon")

Our scatterplot with **ylim** extended to 8 and **col** changed.

curve(0.4\*1.03^x,lwd=2,lty="longdash",col="deepskyblue",add=T)

Our curve is formatted.

**Section 20: Graph texts**

**Rationale:**

We may want to have titles for our figures, change the size of the font of the axes, or change the size of the values along the x and y-axes.

We have formatted points and lines; we can also format text and add more labels. Let’s begin by adding a title to our plot using the argument **main**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time")

Now we will adjust the size of our plot’s text. The default size for everything is 1 but the interpretation of sizes is device-specific, just like it is for **lwd**, so you will likely need to try several different values to find a good size. Setting sizes to values greater than 1 will increase the size while setting the size to values less than 1 will decrease the size.

Let’s start by changing the size of our plot points. The argument is **cex**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5)

Now we can decrease the size of our tick labels using **cex.axis**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8)

Next, increase the size of the axis labels using **cex.lab**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8,cex.lab=1.3)

The size of the title we added earlier can be changed using **cex.main**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8,cex.lab=1.3,cex.main=2)

Our full plot includes not just our scatterplot but also our curve.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8,cex.lab=1.3,cex.main=2)

curve(0.4\*1.03^x,lwd=2,lty="longdash",col="deepskyblue",add=T)

Our audience needs to know what this curve represents. We can add text to our plot using **text**. We need to specify what text to add and the location of the text; size and color can also be indicated. The location of our text is indicated by specifying an (x,y) coordinate, arguments **x** and **y**. The text itself is indicated using the argument **labels**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8,cex.lab=1.3,cex.main=2)

curve(0.4\*1.03^x,lwd=2,lty="longdash",col="deepskyblue",add=T)

text(x=42,y=1.1,labels="y=0.4\*1.03^x")

The text is centered at the point we indicated. We can either adjust the point, i.e. generate new plots gradually changing the **x** and **y** values until the text is in the desired position, or we can modify the location using **pos** to change the position of the text relative to our point and **offset** to fine tune the location of the text. **pos** can have the values 1 to 4: 1 sets the text below our point, 2 sets the text to the left of our point, 3 sets the text above our point, and 4 sets the text to the right of our point. **offset** controls the distance of our text from our point; the default value is 0.5.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8,cex.lab=1.3,cex.main=2)

curve(0.4\*1.03^x,lwd=2,lty="longdash",col="deepskyblue",add=T)

text(x=42,y=1.1,labels="y=0.4\*1.03^x",pos=4,offset=0)

Size and color of the text can be changed using **cex** and **col** within **text**.

plot(faithful$waiting,faithful$eruptions,xlab="Waiting time (min)",ylab="Eruption duration (min)",xlim=c(40,100),ylim=c(0,8),xaxs="i",yaxs="i",pch=16,col="salmon",

main="Eruption duration and waiting time",cex=1.5,cex.axis=0.8,cex.lab=1.3,cex.main=2)

curve(0.4\*1.03^x,lwd=2,lty="longdash",col="deepskyblue",add=T)

text(x=42,y=1.1,labels="y=0.4\*1.03^x",pos=4,offset=0,cex=0.8,col="deepskyblue")

**Afternoon**

**Section 21: Graph axes**

**Rationale:**

We may not be happy with the outputs of our plot, and wish to change the values along a particular axis. This is not a common problem, but it does come up occasionally.

Here is an example of a simple time-series plot.

Time=c(1,2,3,4,5)

Recovery=c(0.5,0.8,1.3,1.9, 2.4)

plot(Time,Recovery)

However, you may be unhappy with the labels of the x-axis. Therefore, let’s change them.

plot(Time,Recovery, xaxt="n")

x=c("Time 1", "Time 2", "Time 3", "Time 4", "Time 5")

axis(1, at=floor(seq(1,5,length=5)), labels=x)

Sometimes you may want to change one of the axes to a log scale. Here is how you would change the y-axis to log scale.

Time=c(1,2,3,4,5)

Recovery=c(0.5,0.8,1.3,19.9, 124.4)

plot(Time,Recovery, log='y', pch=19)

Sometimes we may wish to reverse the order of an axis. Here we reverse the y-axis. It is not that useful in this circumstance, but it can be, for example, if we are dealing with stable isotopes.

plot(Recovery~Time , ylim = rev(range(Recovery)))

In science, we often have complex units. Here we use an expression to show the unit for carbonate production, as kg CaCO3 m-2 y-1.

Site=c(1,2,3,4,5)

Carbonate=c(0.8,1.1,1.4,1.5, 2)

MyYlab = expression(paste("Gross carbonate production ",

"(kg CaCO"[3],

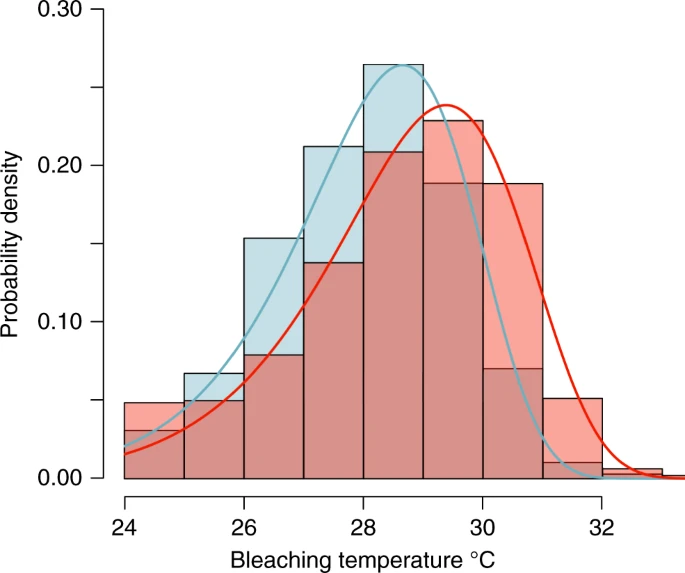
" m"^"-2",

"yr" ^"-1",")"))

plot(Site,Carbonate,ylab=MyYlab,xlab='Site',col='blue')

**Section 22: Histograms**

**Rationale:**

Exploring data using histograms is highly recommended. Sometimes the final product can be a histogram (see figure to the right, in Sully et al. 2019, *Nature Communications*) as a simple histogram can be often more convincing than a complex figure, where the story gets lost. 

Histograms can be plotted using the hist command.

hist(variable\_name)

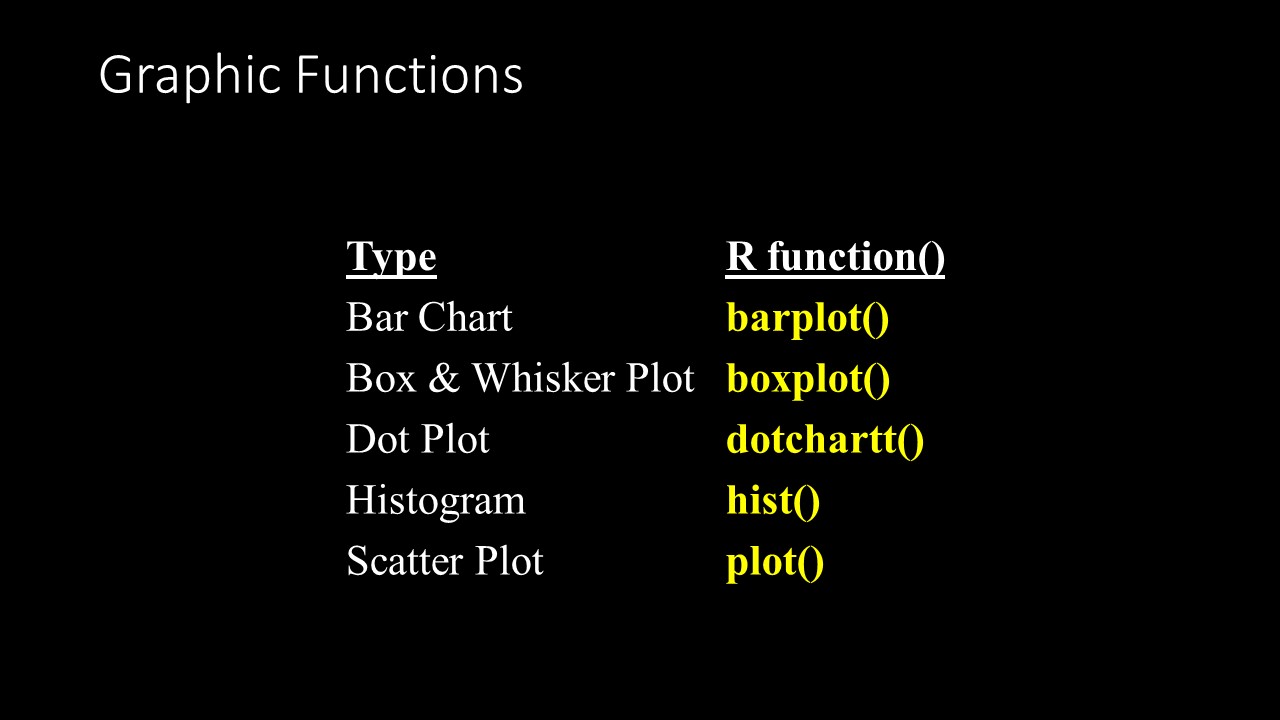
Use ?hist, to find out the features of a histogram plot. Explore the meaning of freq=TRUE and freq=FALSE.

Plot two different histograms, using a data variable of your choice. One of the histograms should have freq=FALSE, and one should have freq=TRUE.

Also plotting the density of a variable can be extremely useful, for example:

plot(density(variable-name))

**Section 23. Boxplots**



Simple **boxplot**

Let’s use the CO2 data again and use a boxplot.

boxplot(uptake~conc, data=CO2)

Many times you may be interested in examining two variables simultaneously, for example, what are the oxygen levels in different tanks with different species. The following is a useful example of plotting a boxplot that illustrates both variables together.

boxplot(uptake~conc\*Treatment, data=CO2)

There are lots of other histogram options:

[https://r-graph-gallery.com/histogram.html](about:blank)

**Day 5**

**Morning**

**Section 24. Bar plots**

**Rationale:**

Categorical predictor variables cannot be plotted using a scatter plot. Bar plots are an effective way to visualize your data when your predictor variable(s) are factors. Additionally, bar plots are easy for your audience to read and interpret.

Bar plots are constructed by first calculating an average or central tendency (i.e. mean or medium) and a dispersion/error (i.e. SD or MAD), saving each as a new object. We learned how to do this earlier using **tapply**. Let’s use the CO2 data again.

mean.uptake=tapply(CO2$uptake,CO2$Treatment,FUN=mean) # mean uptake by treatment

sd.uptake=tapply(CO2$uptake,CO2$Treatment,FUN=sd) # sd uptake by treatment

Now we can begin constructing our bar plot. The bar plot command is **barplot**, and we want to plot the means we calculated above.

barplot(mean.uptake)

Now we need to adjust the y-axis and add a horizontal line across the bottom.

barplot(mean.uptake,ylim=c(0,50))

abline(h=0)

The treatment names are based on the identifiers from the **tapply** command. We can change them using the argument **names**.

barplot(mean.uptake,ylim=c(0,50),names=c("Nonchilled","Chilled"))

abline(h=0)

Now we need to add the error bars for the standard deviations. The command **arrows** will add error bars. We need to specify coordinates from which to draw the arrows and coordinates to which to draw the arrows. The simplest way to do this is to use the bar plot itself as the x-coordinates and a combination of the means and standard deviations as the y-coordinates. Instead of entering the entire plot command, we can save the plot as an object.

co2.treat.plot=barplot(mean.uptake,ylim=c(0,50),names=c("Nonchilled","Chilled"))

Unlike previous times we have saved an object, R will save and display plots.We need to repeat the abline command. The order we will enter the arguments for **arrows** is *plot.object, mean-sd, plot.object, mean+sd*. For this example that becomes: co2.treat.plot, mean.uptake-sd.uptake, co2.treat.plot, mean.uptake+sd.uptake, followed by additional arguments for arrows

arrows(co2.treat.plot,mean.uptake-sd.uptake,co2.treat.plot,mean.uptake+sd.uptake)

The result is an arrow starting at mean - sd and ending at mean + sd but pointing up. We want error bars not an upward pointing arrow. The argument **code** specifies what type of arrow to draw and can take the values 1 (arrow pointing down), 2 (arrow pointing up, the default), or 3 (arrow pointing up and down).

co2.treat.plot=barplot(mean.uptake,ylim=c(0,50),names=c("Nonchilled","Chilled"))

abline(h=0)

arrows(co2.treat.plot,mean.uptake-sd.uptake,co2.treat.plot,mean.uptake+sd.uptake,code=3)

Now we have arrows pointing up and down, but we want bars, not arrows. The argument **angle** will adjust the degree of the angle of the lines of the arrowheads. The default value for **angle** is 30. Use **angle** to make error bars.

co2.treat.plot=barplot(mean.uptake,ylim=c(0,50),names=c("Nonchilled","Chilled"))

abline(h=0)

arrows(co2.treat.plot,mean.uptake-s,co2.treat.plot,mean.uptake+sd.uptake,code=3,

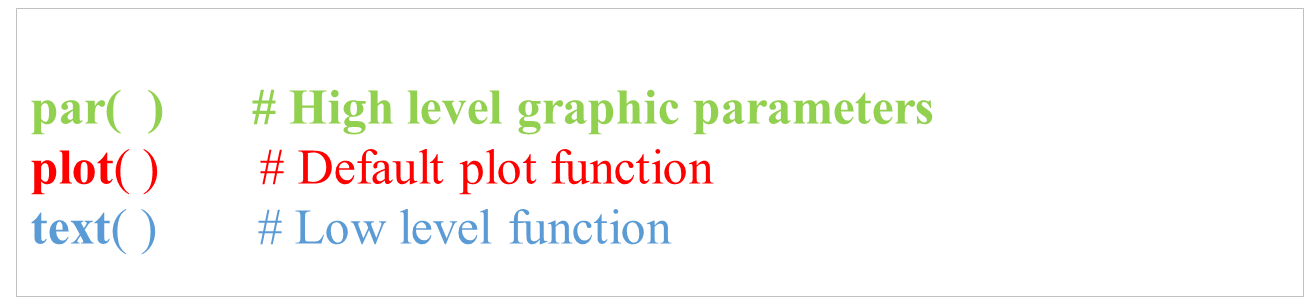
angle=90)

The length of the lines of the arrowheads can be adjusted using **length**. Use **length** to shorten the size of your error bars.

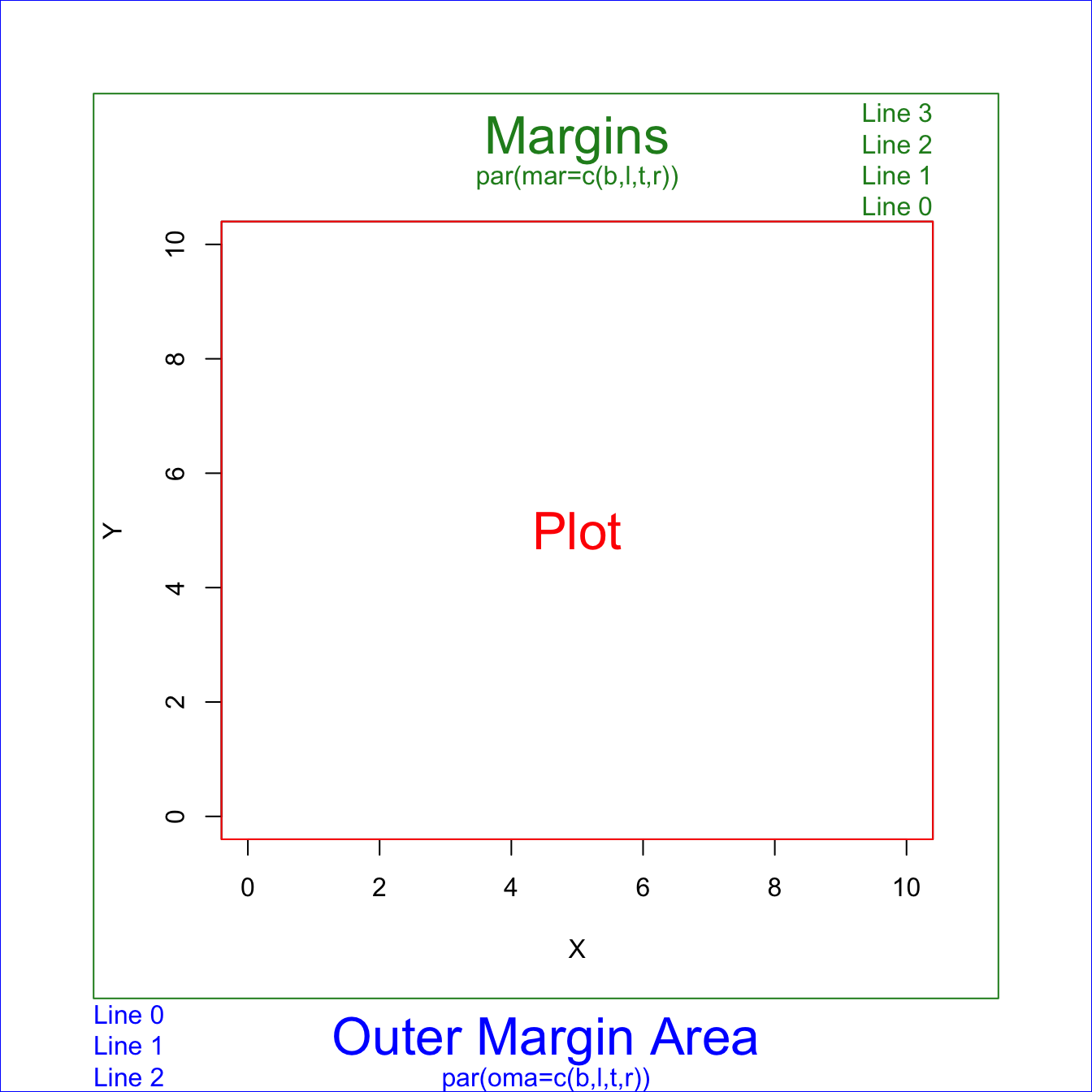
co2.treat.plot=barplot(mean.uptake,ylim=c(0,50),names=c("Nonchilled","Chilled"))

abline(h=0)

arrows(co2.treat.plot,mean.uptake-sd.uptake,co2.treat.plot,mean.uptake+sd.uptake,code=3,angle=90,length=0.1)

**Section 25: Figure sizes**

**Rationale:**

When we wish to publish our results the journals that we publish in have specific requirements for figures. The type of file (i.e., tiff, jpeg, pdf, etc.) and the resolution, or size of the file are also important. 

We may want to have more than one graphic on a panel. In that case, we can partition the graphics panel to give us a framework in which to panel our plots, using par(mfrow = c( nrow, ncol)). For example, we may want to have two figures side-by-side, then we would write before we use the plot command:

par(mfrow=c(1,2)

Or one figure atop another figure, we would use

par(mfrow=c(2,1)

The margins of a figure can be changed using ‘mar’

The order of the arguments is bottom, left, top, and right.

The default is par(mar = c(5, 4, 4, 2))

If you want more space at the bottom of your figure try

par(mar=c(7,4,4,2)

Here is an example of the 3 tiers of plotting (par, plot, and text). This is merely an example as the data has not given.

par(mfrow=c(1,2)) # 1 row, 2 columns;

par(mar=c(8,8,2,1)) # margin lines  
plot(x\_data, y\_data)  
 text(4,10, “My text here”, col = “black” ) # the text is displayed at 4 along the x-axis and 10 along the y-axis

**Section 26: Saving graphs**

**Rationale:**

We may wish to save the figure as a specific file type or at a specific resolution.

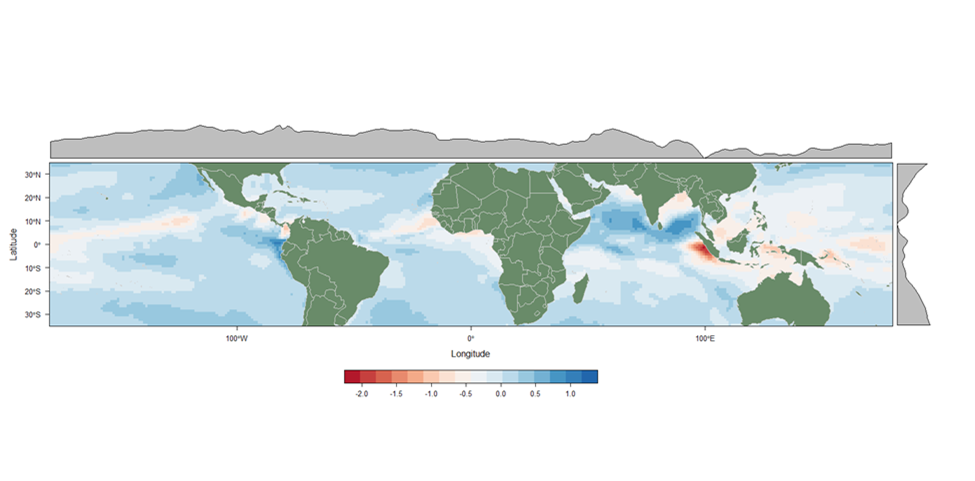
Depending on how we intend on using a plot, we may wish to save the file as a png, jpeg, tiff, bmp, metafile, svg (scalable vector graphics, which is web friendly), postscript(eps), or pdf. Some journals require only high-resolution png files, other journals require pdfs.

jpeg(filename, width = 1800, height = 850, pointsize = 12)

tiff("Plot3.tiff", res = 600)

plot(x, y)

dev.off() #This sets the defaults back, and will remove all the previous plot definitions.



**Afternoon**

**Section 27: Vignettes**



**Rationale:**

Vignettes are short tutorials written by the creators of different packages. They outline features of a package that the authors are particularly proud of. Some packages are better than others, and some vignettes are better than others, but most we have found to be useful.

The best way to access vignettes is to read them into the R environment as a script.

For example, if you are interested in spatial analysis, you should become familiar with the package *sp* and more recently *sf*. One of the vignettes outlines ways to handle shapefiles and manipulate shapefiles. Especially useful in generating new shapefiles where two shapefiles intersect. The vignette ‘over’ outlines the vignette that shows you how to manipulate the shapefiles.

If you are using R, without RStudio, you can generate your script for over, and go through the worked examples, running the code in the script.

install.packages("vegan")

vignette("diversity-vegan")

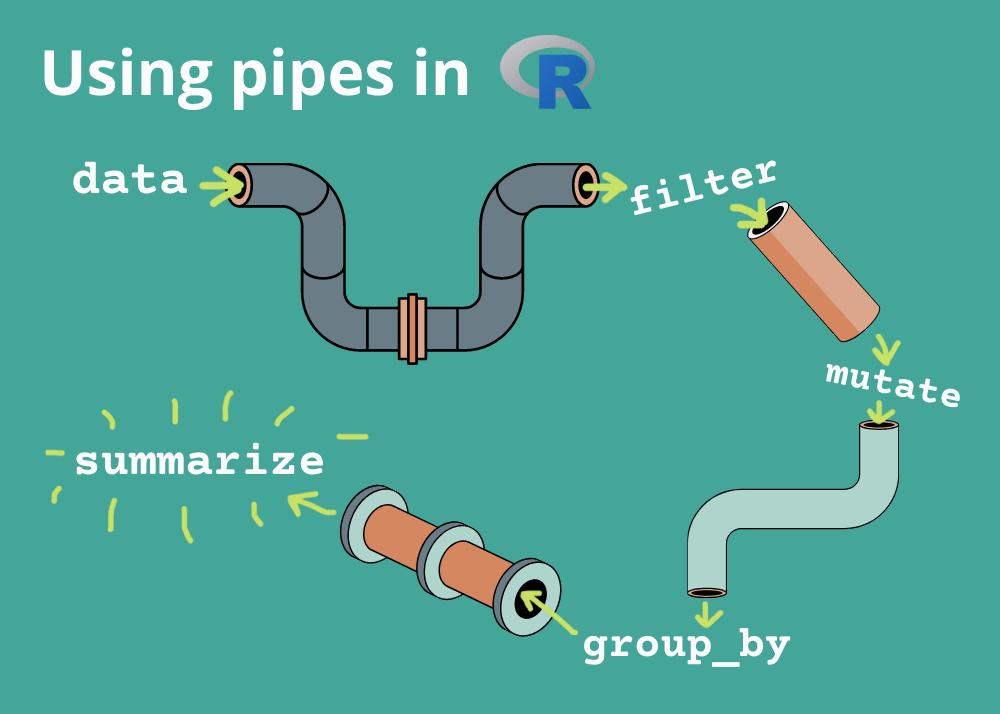
edit(vignette("diversity-vegan"))

If you are using RStudio, then use the following command instead.

library(sp)

edit(vignette("over"), editor = "internal")

**Section 28: Pipes**



**Rationale:**

To reduce the amount of code we can use a concept called pipes.

[https://www.rforecology.com/post/how-to-use-pipes/](about:blank)

This is a pipe %>%

A pipe essentially means ‘then’. Here is an example of a pipe, where we subset **the**n aggregate.

library(dplyr)

data(iris)

iris %>%

subset(Sepal.Length > 5) %>%

aggregate(. ~ Species, ., mean)

For the code above, the **subset()** function filters the dataset to include only rows where the **Sepal.Length** column is greater than 5. Then (%>%) that data is piped into the **aggregate()** function, which groups the data by the **Species** column and calculates the mean of all other columns for each group. The **.** before the tilde (**~**) symbol indicates that all columns should be included in the aggregation. The final result is a new dataset that shows the mean values of all columns for each species where the **Sepal.Length** is greater than 5.