

# Basics of R

## Learning Objectives

- Describe what **R** is
- Interact with R using **RStudio**
  - Become familiar with the key *features in RStudio*
- Learn about common **data types**
- Learn how to create and use **variables**
- Become aware of some basic **built-in functions**
- Gain familiarity installing and loading **packages**
- Try some fundamentals **data wrangling**

## Introduction

This training is the first in a series of tutorials for our 2022 “Bioinformatics Café” starting April 27th, 2022. This is a great opportunity to build the skills needed to succeed in bioinformatics from the ground up. The workshops will be taught by postdocs with years of experience in bioinformatics.

## What is R?

- Free and open-source dynamic and strongly typed programming language
- Provides powerful tools for statistical analysis and visualization
- Important for science, business, education, etc.
  - Usually ranks as **one of the top ten most popular languages**
  - Considered an in-demand skill to learn

## Comprehensive R Archive Network (CRAN)

- CRAN is the central software repository for R, supported by the R Foundation
- Network of ftp and web servers store identical, up-to-date, versions of code and documentation for R
- Hosts many ( $\geq 14000$ ) add-on packages used to extend the functionalities of R

## What is RStudio?

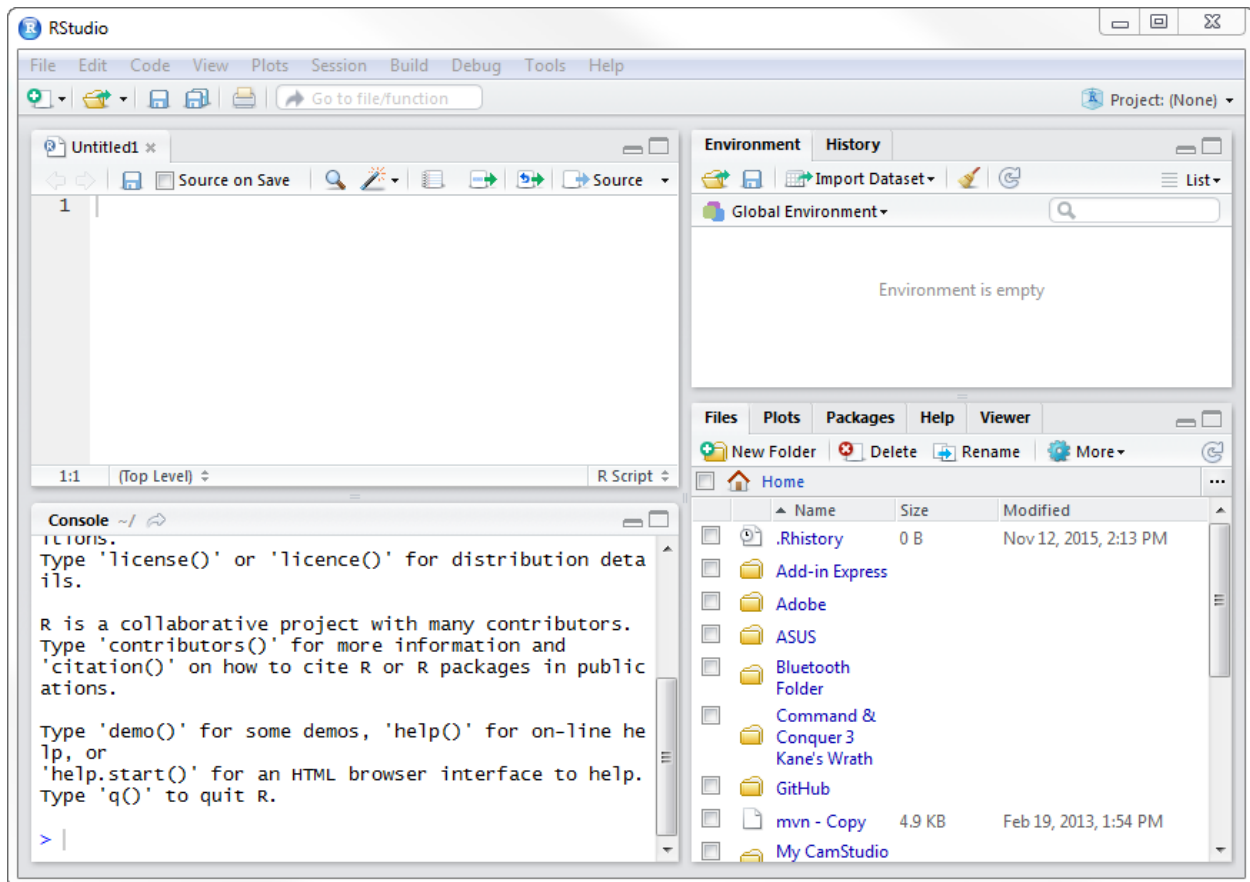
- RStudio is an open source Integrated Development Environment (IDE)
  - Where you write and test code
- RStudio provides an intuitive and feature rich graphical user interface
  - Integrates package management, project management, version control, notebooks, etc.

## Getting started with R and RStudio

- Install R from CRAN
- Select the correct version for your operating system
- Install RStudio Desktop from RStudio

### RStudio Basics

Once you open RStudio, you will see four main panes. Each will contain different information. See the example below.



Starting from the top left pane and going from left to right, we have the descriptions of each pane below:

1. **Source Editor:** This pane is where you can write R scripts or create notebooks. Each new document will have its own tab. Code written here can be executed with the **Run** command.
2. **Environment:** This pane displays objects, variables, and functions that are generated in your R session. There is also a history of all code that was executed.
3. **Console:** This pane is where you can run commands interactively. The output will display in the console.
4. **Files, Plots, Packages, Help, Viewer:** This pane has several tabs that are important.
  - *Files:* This tab shows the structure and content of a directory on your computer. This could be your working directory or a directory that you manually navigated to.
  - *Plots:* This tab will display plots or figures as an output from the console.
  - *Packages:* This tab contains a list of all packages that are installed. Packages that are loaded in your R session will have a checked box.

- *Help*: This tab reveals the help pages for an R package or function.
- *Viewer*: This tab shows compiled R Markdown documents.

## Starting a new project in RStudio

When start a new project with R, it is a good practice to create a specific project directory. This will help you keep your files and data organized by project.

To get started:

1. Open RStudio
2. Go to the File menu and select **New Project**.
3. In the New Project window, choose **New Directory**. Then, choose **New Project**. Name your new directory. You can use a name like, Basics-in-R, and then “Create the project as sub directory of:” in a location of your choice.
4. Click on **Create Project**.
5. The project should open up automatically in Rstudio.

We can view our working directory by using the function `getwd()`

```
getwd()
```

Your working directory will be the location where R will automatically look for files. If you want to find files in a different location, you will either need to provide the full path or type the path in relation to the working directory. Files that you output will automatically save into your working directory unless a path is provided.

To organize your working directory, it is highly recommended to generate sub-folders like **data/** or **results/**. You can do so in the Files tab and select **New Folder**.

## Basic math operations in R

Below are some basic math operations available in R.

Operation	Symbol
Addition	$a + b$
Subtraction	$a - b$
Multiplication	$a * b$
Division	$a / b$
Exponent	$a ^ b$
Remainder	$a \% b$
Integer Division	$a \%\% b$

For instance, here is an example of addition.

```
5 + 3
```

---

## Exercise

1. Use R as a calculator and find the square root of 10
- 

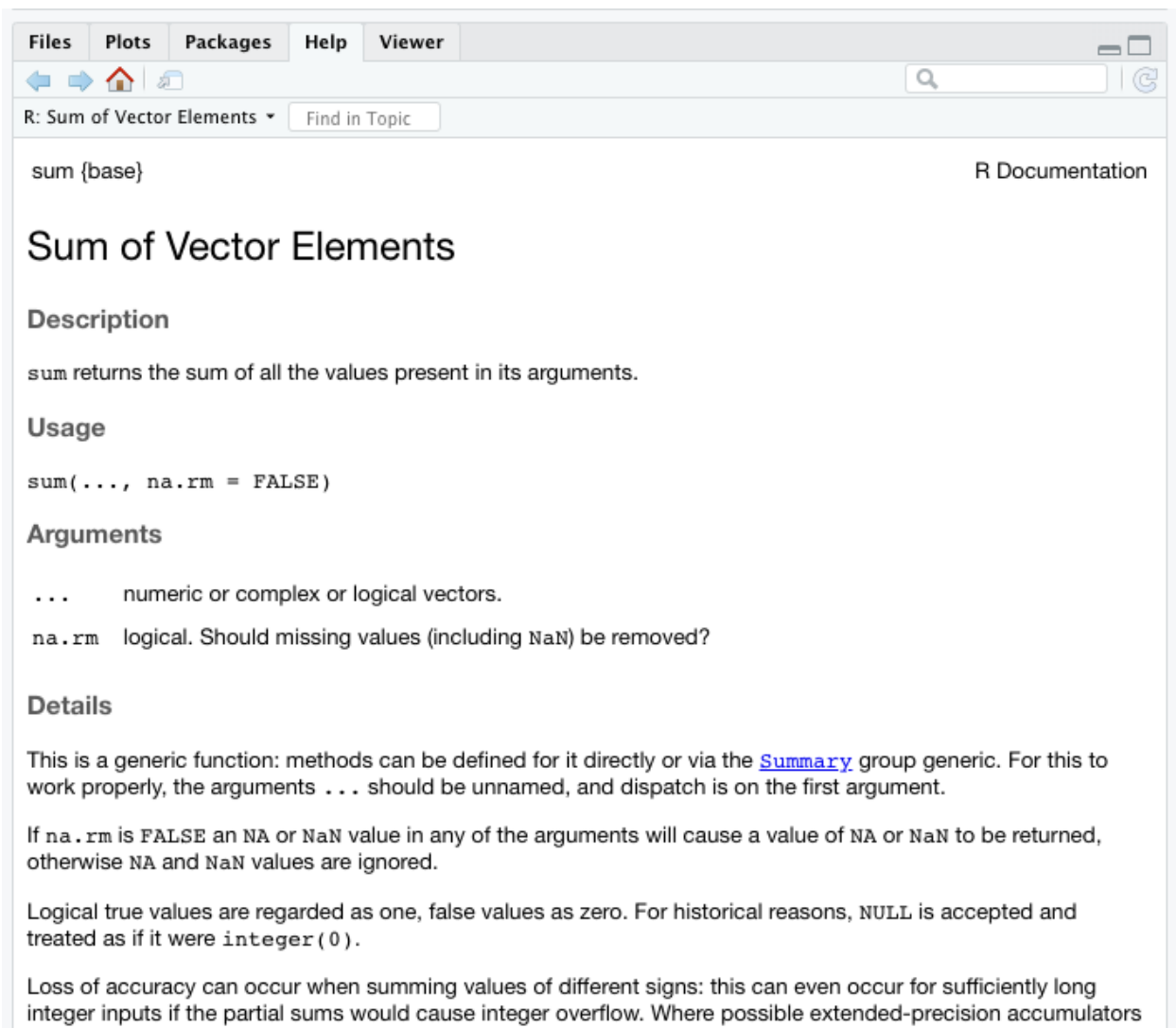
## Built-In Functions in R

R has several pre-built functions. For instance, we can use `sum()` instead of the `+` symbol for addition.

```
sum(1,3) #This gives the sum of 1 and 3
```

We can also call the *R documentation* for a function by using `?` before the function name. The documentation will show up under the Help tab. It contains information regarding description, usage, and arguments for a function.

```
?sum #Find the R Documentation for sum()
```



The screenshot shows the R Help window with the 'Help' tab selected. The title bar indicates 'R: Sum of Vector Elements'. The main content area displays the documentation for the `sum` function from the `base` package. The documentation includes sections for Description, Usage, Arguments, and Details. The Description states that `sum` returns the sum of all values in its arguments. The Usage shows the function signature `sum(..., na.rm = FALSE)`. The Arguments section explains that `...` represents numeric, complex, or logical vectors, and `na.rm` is a logical flag to remove missing values. The Details section provides additional information about the function's behavior, including its generic nature, handling of `NA` and `NaN` values, and potential loss of accuracy with large integers.

sum {base} R Documentation

## Sum of Vector Elements

### Description

`sum` returns the sum of all the values present in its arguments.

### Usage

```
sum(..., na.rm = FALSE)
```

### Arguments

`...` numeric or complex or logical vectors.

`na.rm` logical. Should missing values (including `NaN`) be removed?

### Details

This is a generic function: methods can be defined for it directly or via the [Summary](#) group generic. For this to work properly, the arguments `...` should be unnamed, and dispatch is on the first argument.

If `na.rm` is `FALSE` an `NA` or `NaN` value in any of the arguments will cause a value of `NA` or `NaN` to be returned, otherwise `NA` and `NaN` values are ignored.

Logical true values are regarded as one, false values as zero. For historical reasons, `NULL` is accepted and treated as if it were `integer(0)`.

Loss of accuracy can occur when summing values of different signs: this can even occur for sufficiently long integer inputs if the partial sums would cause integer overflow. Where possible extended-precision accumulators

## Exercise

1. Use `sum()` to add the numbers 10, 20, 30, 40, and 50.
- 

## Math functions in R

In addition to the basic calculations, there are many common pre-built math functions in R.

Operation	Function
Square root	<code>sqrt()</code>
Logarithm	<code>log()</code>
Logarithm, base 10	<code>log10()</code>
Exponential	<code>exp()</code>
Summation	<code>sum()</code>
Round	<code>round()</code>
Mean	<code>mean()</code>
Median	<code>median()</code>
Minimum	<code>min()</code>
Maximum	<code>max()</code>

```
#What is the square root of 10?  
sqrt(10)
```

---

## Exercise

1. Find the round the summation of 10.13 and 53.535 to one decimal place.
  2. Hint: try `?round` for help.
- 

## Defining variables

Variables are fundamental for building reproducible and intuitive programs. They help us store data in the computer memory that becomes *callable* or accessible in different parts of a program. As you will see, variables also allow us to reduce the amount of typing. R has two *assignment operators* for defining variables: `<-` and `=`. The operator `<-` can be used anywhere, whereas the operator `=` is only allowed at the top level.

```
x <- 1  
y <- 15.3
```

Let's add `x` and `y`.

```
#What is in x + y?  
x + y
```

Importantly, when defining variable names, ensure that you use an *informative* name. This enables yourself and others when reviewing code to know how the variable was used. For instance, we used `x` and `y` in the examples, but their meaning is unknown. Something like `country_population` or `room_capacity` provides better definition for a variable.

## R Data Types

There are different types of data in R, which can be stored as a variable. Below is a table of some of the most commonly used data types.

Data Type	Definition	Example
numeric	Any number value	3.14
integer	Any whole number value	42
character	Any number of ASCII characters defined within quotation marks	"Hello world!"
logical	A value of TRUE or FALSE	TRUE
factor	A categorical type of data	#> [1] Male Male Male Female Female #> Levels: Male Female

The function `class()` can be used to find out the type of data that you are dealing with.

```
x <- 3
class(x)
```

```
x <- TRUE
class(x)
```

## Relational and Logical Operators

In R, there are relational operators that compare values between two variables. Typically, these are numerical equalities or inequalities. The result of comparison is a Boolean value.

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to
%in%	is 'in' a given vector

```
#Is 3 greater than 5?
3 > 5
```

There are also logical operators which connect two or more expressions depending on the meaning of the operator. These are typically combined with relational operators.

Operator	Description
	OR
&	AND
!	NOT

```
#Is 3 greater than 1 and is 3 greater than 5?
3 > 1 & 3 > 5 # True

#Is 3 greater than 1 or is 3 greater than 5?
3 > 1 | 3 > 5 # True or False = True

#The NOT operator switches the value of a boolean
print(!TRUE) # this is printing the output of the operation NOT(TRUE) = FALSE
print(!FALSE) # this is printing the output of the operation NOT(FALSE) = TRUE
```

## Exercise

Find the answer to  $10 < 0$  and  $1 < 2$

```
# Write your code here
```

## R Data Objects

### Vectors

Vectors are a data structure in R containing one or more values. In fact, you may have noticed a `[1]` in the output of `x`. This indicates that it is a vector of length 1.

```
length(x) #This function gives you the length of a vector
```

We use the function `c()` to define a vector with multiple elements. The `c` stands for combine.

```
my_first_vector <- c(1,2,3,4,5) #We can also do this with the following, 1:5 instead of c()
my_first_vector
```

We can add more elements to the same vector.

```
my_first_vector <- c(my_first_vector, 6,7)
my_first_vector
```

We can select (or *index*) a subset of elements in a vector. We do this by defining which position we want in brackets `[]` after the vector. **Note:** In R the first index is 1. This can be different in other languages. For example, the first index in Python is 0. For example, to get the first index in R you write: `my_first_vector[1]`. The equivalent Python code is `my_first_vector[0]`.

```
#Let's take out the 3rd element
my_first_vector[3]
```

What if we wanted to select multiple elements? We can use another vector with the positions we want.

```
#Let's take out the 2nd and 4th elements
my_first_vector[c(2,4)]
```

We can also select a set of elements using a colon. For example, to get elements 1 to 3

```
my_first_vector[1:3]
```

We can select all elements but a single or multiple element by using `-`.

```
#Let's keep all but the 5th element
my_first_vector[-5]
```

```
#Let's keep all but the 1st and 3rd elements
my_first_vector[-c(1,3)]
```

Importantly, R functions are typically *vectorized*. This means that the function will perform its operation on all elements of the vector without having to loop through for each element.

```
my_first_vector
my_first_vector * 2
```

We can also test some of math functions we listed above.

```
mean(my_first_vector) #This gives the mean of a numeric vector
min(my_first_vector) #This gives the minimum numeric value in a vector
max(my_first_vector) #This gives the maximum numeric value in a vector
```

## Matrices

Matrices are second ranked tensors (elements are represented with two indices) whereas vectors are first ranked tensors (you only need one index to specify the elements). Matrices have a dimension attribute that tell us the number of rows and columns.

The `matrix()` function is used to create a matrix. We provide the values, `#` of rows and `#` of columns.

```
#matrix of sequence 1 to 6 with two rows and three columns
my_matrix <- matrix(1:6, nrow = 2, ncol = 3)
print(my_matrix)
```

```
dim(my_matrix)
```

```
my_matrix[1,2] #first row and second column
```

Matrices can be created by adding vectors using `cbind()`

```
vector_1 <- c(1,2,3,4)
vector_2 <- c(2,3,4,5)

matrix_1 <- cbind(vector_1, vector_2)
print(matrix_1)
```



---

## Exercise

1. Create a matrix containing a sequence from 1 to 10 with 5 rows and 2 columns
  2. Add a row vector of values `c(3,5)` to the matrix
- 

## List

Lists are objects that can contain elements of different classes (strings, integers,...)

Lists can be explicitly created using the `list()` function.

```
my_list <- list('I love science!', 1, TRUE)
print(my_list)
```

Like vectors, we can use indexing to select specific elements of a list. Using single brackets `[]` will create a new list with the index or indices selected.

```
my_list[1:2]
```

Using double brackets `[[ ]]` will not create a new list and only selects the element.

```
my_list[[1]]
```

---

## Exercise

1. Can we do `my_list[1,2]`?
  2. Can we do `my_list[[2:3]]`?
- 

## Data Frames

Data frames can be used for tabular data in R. They are similar to an excel worksheet. Unlike matrices, data frames can store different classes of objects in each column.

Data frames can be created using the `data.frame()` function.

```
my_dataframe <- data.frame(A = 1:5, B = c(T, T, F, F, T))
print(my_dataframe)
```

## Installing Packages

Although R comes with many pre-loaded packages, installing new packages gives you access to additional functions. Packages can be downloaded from CRAN, Github, or Bioconductor. Some functions available in new packages can reduce complex tasks to a single lines of code.

## CRAN

CRAN is the official repository for R packages. To install a package, use `install.packages("name_of_package")`. The package name in quotations is the argument for this function. For example, we can install the data visualization package `ggplot2`, which we will use later.

```
install.packages("ggplot2")
```

We can load a package by using the `library()` function. To load the `ggplot2`, we do the following:

```
library(ggplot2)
```

## Other Repositories

- Github, which is an online software repository provider, hosts several repositories of open-sources R packages. There are several R packages that be can be found in early phases of development that you can find and test out. Github is not limited to R and software written in other languages can be found there too. Oftentimes once the software has completed development or has reached a major version, it may move to CRAN or Bioconductor.
- Bioconductor is a repository that focuses on R packages for biological assays. They host packages for analysis of microarrays, RNA-seq, ATAC-seq, scRNA-seq, among many more.

## Working with Data

A major reason many use R is to analyze data in a reproducible manner. In that case, we need to be able to save data and also read in data from other sources. We can read in data through several means.

### Reading in data

There are several methods for reading in tabular data.

- `read.csv()`
- `read.table()`
- `read.delim()`

All of the above function perform similarly in that they import tabular data such as those in `.csv` or `.txt` format.

```
iris <- read.csv(file = 'iris_dataset.csv')
```

### Exploring your data

The data that we imported is the `Iris` data set. It is a well-known data set of classifying iris plants. Let's see what type of data we imported by using the `class()` function.

```
class(iris)
```

We use `dim()` to see how many rows and columns the data has.

```
dim(iris) #find rows and columns
```

To explore the first 6 row, we can use the `head()` function. This may be helpful when analyzing large data sets.

```
head(iris)
```

There are 4 columns that are made up of numbers and one with strings. The fifth column **Species** is the classification.

---

## Exercise

1. How can we look at more than 6 rows when using `head()`?

---

This tells us that there are 150 rows and 5 columns in this data frame. It is also good practice to use the `str` function to briefly look at the structure of the data.

```
str(iris)
```

Again, this confirmed our brief look earlier that there are 4 columns of numeric values. Using `summary()`, we can acquire some insight on the data distribution.

```
summary(iris)
```

We can start by looking at the data point distribution of `Sepal.Length` between the three species. We use the library `ggplot2` and create a scatter plot with `geom_point()`

```
ggplot(iris, mapping = aes(y = Sepal.Length, x = Species, color = Species)) +  
  geom_point()
```

## Saving data

When working with R, you may accumulate several processed data in the Environment pane that you want to save and load at another time. In addition, you may want to send this data to another system that has R.

If you are keeping good practices, you would have maintained a notebook and save all of your code. In that case, you could re-run all the code, but it will take time and computation power.

Saving the data as an R data file can save time.

There are a few ways to save your data.

We can use `saveRDS()` to compress a single object as an `.rds`.

```
saveRDS(iris, file = "my_data_iris.rds")
```

To load the data, use the `readRDS()` function. You need to have the path to the file.

```
readRDS(file = "my_data_iris.rds")
```

Using the `save()` function, we can save multiple files as an `.Rdata`.

```
# Save multiple objects
save(iris, my_matrix, file = "iris_my_matrix.RData")
```

To load the data again, use the `load()` function.

```
load("iris_my_matrix.RData")
```

Finally, if you want to save your entire workspace and this may include various objects that are not part of the analyses. We use `save.image()`.

```
save.image(file = "my_work_space.RData")
```

The workspace can be loaded into R using `load()`.

```
load("my_work_space.RData")
```

## Best Practices

1. Document your code using descriptive comments.
  - This will help you and others understand what you did in the future.
2. Keep ideas and explanations for your code in `.Rmd` or `.md` files.
3. Create and work inside a directory specific to your R project.
  - This helps with code and data organization.
4. Use intuitive names for variables and avoid vague names like `x` and `y`.
5. When in doubt, use the abundant resources online ( $> 2 \times 10^6$  users)
6. Practice and keep learning! *"Make mistakes faster"* -AG

## Additional Resources

- Introduction to R and Rstudio: Harvard Bioinformatics
- Introduction to R and RStudio: Alex Lemonade Stand
- Introduction to R and RStudio: Yale CRC
- R Software Handbook
- R Programming for Data Science