Amidst Lines and Circles: A Seminar on the Do's and Dont's of PCA and other Statistical Techniques

A guide to R syntax and the use of R for mathematics and statistics

Lloyd A. Courtenay[1,2,*]

[1]CNRS, PACEA UMR 5199, Université de Bordeaux, Bât B2, Allée Geoffroy Saint Hilaire, CS50023, Pessac 33600, France
[2]Deparment d'História de l'Art, Universitat Rovira i Virgili, Avignuda de Catalunya 35, 43002, Tarragona, Spain
[*]Corresponding Author Email: ladc1995@gmail.com

This document provides an introduction to the R programming language and its basic syntax, outlining the fundamental building blocks needed to work with R. It covers essential elements of R syntax and demonstrates how to program basic structures, such as functions, conditional statements, and loops. Towards the end, I have included examples that illustrate how to approach mathematics from a more programmatic perspective. These examples can serve as a useful guide for introducing readers to R as a powerful tool for mathematical computation, statistical analysis, and simulation. This initiative has been supported and led by the EVODIBIO team, and SURF-ACE3 working group, of the PACEA laboratory, University of Bordeaux. For this reason I'd like to thank the great support of Antoine Souron, Adeline Le Cabec, Alain Queffelec, and Luc Doyon.
This document was written from the perspective of a programmer, not an archaeologist. The reason I have done this is I wanted to provide the most amount of detail and theory possible for those who are interested, yet have structured the document in a way that if you do not wish to know all the ins-and-outs of programming, you can go directly to the part that interests you the most. However, the goal here is to try and provide a foundation for any user who wishes to dive deeper into the world of computational sciences, or serve as a guide to new users regarding the basic syntax. While I hope this document is as clear as possible to any user, beginner or experienced, if it isn't please let me know, my email is provided above. I will update this document as time progresses to try and create the best guide possible. Nevertheless, to provide a document that is more user-friendly to archaeologists, I have tried to include some examples that might be more accesible to our field.
This first version of the document was written on the 16th of May, 2025, adapted from lecture notes in Spanish I gave to masters students in the geomatics engineering masters degree of the University of Salamanca, between the years 2020 and 2023, on programming theory, GIS and spatial statistics.

Contents

An introduction to R and RStudio **4**
What is a programming language? . . . 4
An introduction to R . . . 5
Installing R . . . 5
Installing RStudio . . . 6
An Introduction to RStudio **6**
Working with R in RStudio . . . 7
Working with libraries in R . . . 9
Basic R Syntax **10**
Comments . . . 10
Types of Variables . . . 10
Basic Data Structures . . . 12
Vectors . . . 12
Matrices . . . 12
Arrays . . . 12

The : operator and sequences . . . 14
Declaring variables and storing them in memory . . . 14
Mathematical operations . . . 14
Logical Operators (&, |, !, =, < and >) . . . 16
Special Values in R . . . 17
Inf and -Inf . . . 17
NaN . . . 17
NA . . . 18
NULL . . . 18
Special Operators in R . . . 19
The :: operator . . . 19
The %in% operator . . . 19
The |> operator . . . 20
The %<% operator from magrittr . . . 20
The + operator from ggplot2 . . . 20
Complex Data Structures . . . 21
Lists . . . 21
Data Frames . . . 22
Indexing and using $, [ ] and [[ ]] correctly . . . 22
Indexing a Vector . . . 23
Indexing a Matrix . . . 23
Indexing an Array . . . 24
Indexing a List or Data Frame . . . 25
Conditional Statements . . . 26
Loops . . . 28
for Loops . . . 28
while Loops . . . 30
repeat Loops and the break function . . . 30
Functions and Object Oriented Programming **31**
The Function-Oriented Programming Paradigm . . . 31
The Object-Oriented Programming Paradigm . . . 31
Functions . . . 31
Object Oriented Programming . . . 36
S3 Classes . . . 36
S4 Classes . . . 36
Approaching Mathematics Programmatically **36**
A quick first example . . . 37
Another quick example - the Euclidean distance . . . 38
A dictionary of common mathematical symbols, and how to program them . . . 39
Vector notation . . . 39
Matrix notation . . . 39
Matrix Operations . . . 40
f(x) - the mathematical function . . . 40
Summation . . . 40
Product . . . 40
Derivatives . . . 41
Integrals . . . 41
Piecewise Functions . . . 42
Trigonometry . . . 43
Greek Letters . . . 44
Other symbols and operators . . . 44

An introduction to R and RStudio

What is a programming language?

A programming language is a formal system used to communicate instructions to a computer. It allows humans to write commands that the computer can interpret and execute to perform a specific task. Like any language, programming languages have syntax and semantics that define how they are written and what they mean. Beyond these rules, there are also stylistic conventions, best practices, and schools of thought that influence how code is written in the field of computational sciences.

Different languages are designed with different purposes in mind, which often leads to the common question: "What is the best programming language?" A more helpful version of this question might be "What is the best programming language *for me*?" — but even that oversimplifies the issue. The right language often depends on the context, goals, and background of the programmer.

There are many types of programming languages, which are often categorized by their "level." To illustrate this, imagine an onion: at its center is the core, and around it are layers stacked neatly on top of one another. Programming languages can be understood in a similar way.

At the core of this metaphorical onion lies machine language—the lowest level of programming. This consists of binary code: literal sequences of 0s and 1s that directly correspond to instructions a computer can execute. On top of this are progressively more abstract languages that resemble human language more closely. These are called high-level programming languages, and they are designed to be easier for humans to read and write. They allow programmers to express instructions using natural language elements and abstract ideas, without needing to manage low-level details like memory addresses or processor instructions.

In this layered view, programming languages are stacked on top of one another. This abstraction is fundamental to how we interact with computers: we write instructions in a human-readable language, which are then translated—step by step—into lower-level representations, until they become binary code the computer can run.

Consider the following example in a high-level language;

```
print("Hello!")
```

The word **print** is familiar — we're literally asking the computer to display the word "Hello!" on the screen. But beneath this simple line, much more is happening. That command eventually gets translated into many more detailed operations handled by the computer.

Compare it with the equivalent in C++:

```
int main() {

    std::cout << "Hello!" << std::endl;

    return 0;

}
```

As you can see, this version is already more verbose and requires understanding of syntax like `main`, `std::cout`, and `std::endl`. This demonstrates the benefit of abstraction: high-level languages make it easier to write code for a wide range of tasks without worrying about all the underlying machinery.

While you don't need to know low-level languages like C++ to begin programming, having a basic understanding of what's happening "under the hood" helps. It can make code more efficient, help with things like debugging, and give you insight into how your instructions are being translated for the machine. After all, programming is about communication, and just like in human languages, things can get lost in translation—especially if we don't follow the rules clearly. Likewise, regardless of what some might say, programming languages are often **still a black box**. There are a lot of things going on, that most people don't truly read in to, but knowing how things work can help us try and become more transparent researchers, and help us identify potential issues with what we are doing.

Before we go on to the specific language of R, however, it is fundamental that we also clarify what the idea of *clean code* is. Writing clean code means writing in a way that is clear, consistent, and easy to understand—not just for ourselves, but for others who may need to read, review, or build upon our work. This includes collaborators on a project, future students, peer reviewers, or even our future selves revisiting old scripts. In scientific and data-driven fields, where code often forms the backbone of reproducible research, clarity is not just a matter of style—it is a responsibility. Clean code improves transparency, reduces errors, and makes it easier to communicate our ideas effectively, whether in publications, presentations, or shared projects.

For those that are interested, I strongly recommend the book *Clean Code: A Handbook of Agile Software Craftsmanship*, by R.C. Martin (2009)[1]. While this book was written for users of Java, which is a much lower level programming language than we are used to, it clearly explains some very important concepts that will help us as researchers. A lot of the rules that apply to Java, however, are too strict for languages such as R and Python. Nevertheless, underneath I highlight some of my favourite

points, that are truly useful for everyone;

1. **Use clear and meaningful names for variables, functions, classes and methods**. Names should be relevant, making it obvious what the code is doing. An ambiguous variable called "a", for example, or "b", tells us nothing about what it is doing. Even "PCA" can be confusing to some users. It is better to write out longer names, that clearly state what they are for, than cryptic codes that only a single user can understand. I personally only use names like "a" or "i" in the extreme cases where I am literally transliterating a mathematical formula, where *a* or *i* are defined mathematically by what I am trying to implement.

2. **Write code for humans, not machines**. Code is read more than it is written, so we should keep the future reader in mind when programming our software.

3. **Use comments only when necessary!** This point is one that surprises many, and some people strongly disagree with. However, as described by Martin (2009), if your code is clean, then you shouldn't need a comment to explain what it is doing. If it is not obvious what you are doing, then the code should be cleaned! Comments should only explain the *why* factor, not the *what* (why am I doing what I am doing, as opposed to the what I am doing in the first place).

4. **Functions should be small, and do only one thing.** This is one of the points that is much harder for R or Python users, however can often be a valuable point to follow if we take it with a pinch of salt. The purpose of a function, as will be described later, is to avoid duplicate code. We want to keep our code as simple as possible, so that finding issues is much easier. Therefore having enormous functions makes little sense and can often be harder to read, maintain, and even use. For this point, I suggest alternative wording; *Functions should be small, and do a limited handful of useful things*.

While I myself have not followed many of these rules in the past, and my earlier codes certainly do not fulfill these criteria (I simply had never heard of them at this point), as I have learned them and gained more experience as a programmer, I have noticed an enormous change over time with the efficiency, accuracy and usability of my code. This makes me feel not only like a better programmer, but a better scientist and researcher in general; cleanliness is certainly a fundamental component of transparent research!

An introduction to R

R is a free programming language, founded in 1993, and one of the most widely used languages among statisticians. It is a high-level language, built on a foundation of C, Fortran, and R itself. Among its competitors, we find other free programming languages with similar uses, such as Python and Julia. From a more commercial perspective, R's proprietary competitors include MATLAB, SAS, and SPSS.

Today, R is considered one of the most popular programming languages from multiple perspectives, and it is used across a wide range of fields—not only for developing statistical applications but also for data mining, text mining, and algorithm design in the field of artificial intelligence.

The R project also has its own peer-reviewed, open-access, online scientific journal. The R Journal has been indexed in the Journal Citation Reports (JCR) since 2012, with an impact factor of 3.312 in 2019. It has been classified as a Q1 journal in the field of statistics and probability since 2018, and as a Q2 journal in computer science since 2018.

Although R has its own command-line interface, we can take advantage of various tools available for designing statistical projects. For this, we will primarily work in the integrated development environment (IDE) RStudio.

Installing R

At the time of writing this guide, the most up-to-date version of the R language is version 4.5.0, released in April 2025. R can be easily and freely downloaded from its official website.

If you are working on Windows, installing R can be as simple as searching for "Download R" on Google. The first link will take you directly to the relevant page.

If you click on the link "Download R 4.5.0 for Windows," your browser will download the .exe file to install R without any complications.

For Mac or Linux users, you can download R by visiting the following website: `https://www.r-project.org/about.html`. On the left side of the page, you will find several navigation links. Click on the link labeled "CRAN" (Comprehensive R Archive Network) and select the country that corresponds to you (or the one closest to you). CRAN is a network of servers that allows access to various mirrors, facilitating the download and management of libraries for R.

Click on the link that corresponds to your operating system. In the case of Windows, this will lead you to a simple page with several subdirectories. Clicking on the link named "install R for the first time" will bring you back to the page mentioned earlier in this section. For Mac users, the CRAN page will take you to a page with different links. From here, you can click directly on the first link under "Latest Releases" » "R-4.5.0.pkg" to download the .pkg installer for R.

For Linux users, the following link provides straightforward instructions to download and install R: https://www.r-bloggers.com/2013/03/download-and-install-r-in-ubuntu/

Once you have downloaded the R installer, run it and follow the instructions until the installation is complete. You will then have several new icons on your desktop to launch R in either 32-bit or 64-bit mode.

If you want to verify that R has been installed correctly, open R (NOTE: my operating system is 64-bit), and you can start typing code directly in the console that appears as soon as the program opens.

A very strong recommendation that I would like to include here is the value of also installing the RTools extension; https://cran.r-project.org/bin/windows/Rtools/. RTools provides the necessary components for the compilation, reading and building of many components of R, and is fundamental if you work in developing your own R libraries. In addition, quite a lot of R libraries use dependencies from RTools, so it is very valuable to have installed straight away. Simply download the latest stable version that corresponds to your version of R, and follow the instructions. The most recent releases of RTools will already add it to your computer's path, however in the past this was not the case. If you are using an old version of R you will need to check the specific steps required to ensure that RTools is installed on your computer.

To check that everything is working, open up the R Console, and type in the following;

```
install.packages("devtools")
```

This is the first basic function you will learn, necessary to install libraries and other components that will help us in our analyses. The first time you run an "install.packages" command, you will likely be asked to select a CRAN mirror. If this is the case, simply select the server closest to you. Once devtools has finished installing, you can check that both RTools and R have been installed correctly by typing:

```
library("devtools")
find_rtools()
```

If this results in the word "TRUE" being printed on the screen, you have successfully done everything to start with the best tools available for R.

Installing RStudio

You can download RStudio from its official website: https://rstudio.com/

RStudio offers a variety of products, but we will be working with the most basic version. To access it, click on the "Products" menu and select the first option that appears.

Next, look for the version called RStudio Desktop, click on the link "Download RStudio Desktop," and select the first option: RStudio Desktop Free. Depending on your operating system, follow the instructions that appear on screen to first download and then install RStudio.

An Introduction to RStudio

RStudio is an integrated development environment (IDE), designed specifically for working with the R programming language, but it can also function as a text editor with tools for working in Python, SQL, JavaScript, C++, and even integrating these languages within a single application. It is common to find RStudio cited as the "software" used to do analyses - while this is not necessarily wrong, it does require some clarification and there are better ways of expressing the same concept. R is a programming language that can be used to write software. In itself, therefore, R is not the software, but is what is used to create the software. RStudio, on the other hand, is more of a text editor, that allows us to work more efficiently with R, and interact more naturally with the programming language itself. RStudio offers a bunch of advantagous tools such as colour syntax, built in plot, debugging and file management tools, while also being fast and diverse. However, you are not obliged to use RStudio to work in R, R can just as well be written in another text editor, such as Visual Studio Code, for example.

The first time we open RStudio, we should see three main windows (typically):

On the left, we have the **Console** — this is where we can run R code, just like in the standard R console. Depending on your version of RStudio, you may also have access to the system terminal (older versions don't always include it). The Terminal is equivalent to the command prompt (e.g., cmd.exe on Windows), and allows us to run Python code (for example), or interact with the operating system more generally. For most purposes, however, we will mainly use the R console.

On the right, there are two main panes. The top-right window has several tabs that let us view the contents of the **environment** (loaded tables, defined functions, etc.), the **history** (recently executed code), and a **Connections** tab for working with external servers.

The bottom-right window contains tabs for navigating your computer's **directories**, viewing **plots** and figures that we generate, **managing packages**, accessing **R documentation**, and working with **web-related tools** (like JavaScript visualizations, HTML, or widgets).

The top-right window is known as the **Environment**, and the bottom-right window is known as the **Viewer**.

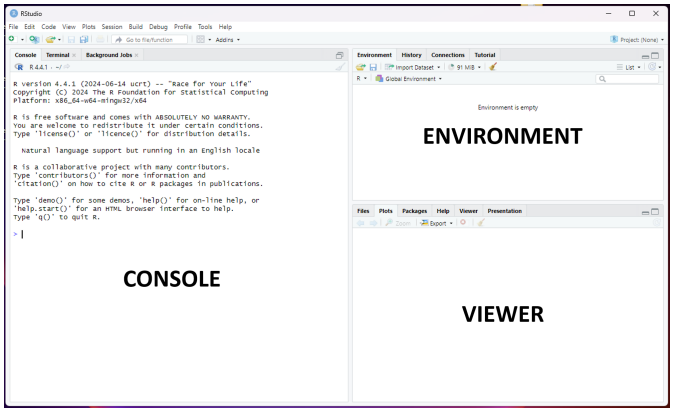

Figure 1. An example of the RStudio screen

Technically there is a 4th window that is currently hidden, however we will begin by setting up our work environment before getting to that part of the interface.

There are a bunch of things that RStudio helps us do that makes life much easier, and that stems in the idea of working in *RStudio projects*. A project allows us to localise all of our code, files and folders in one place that is easily accessible to RStudio. For example, typically to open a file, we would have to type the entire route of the folder where the file is located, however, if we work in a project environment instead, R already knows that the majority of the files we will be working with are located in the same folder to wherever the project file is.

The way to create a project is to go into the **File** menu, and select **New Project**. This will open up an external window asking whether you would like to create a new directory, or an existing directory. If you already have a folder where all of your data is, simply select existing directory, and specify where in the computer your folder is. Once you have created the project, in your folder you should see a file with the extension **.Rproj**. This is simply a file that contains all of the metadata of your project, if you open this file, it will open up R in the context of your project, with everything you had been working on up to this point ready to go. Finally, you will also notice that at the top of the console, next to the version of R, there is a small text indicating the path of where the directory is loaded, succesfully telling us that R already knows that everything it needs is in that folder.

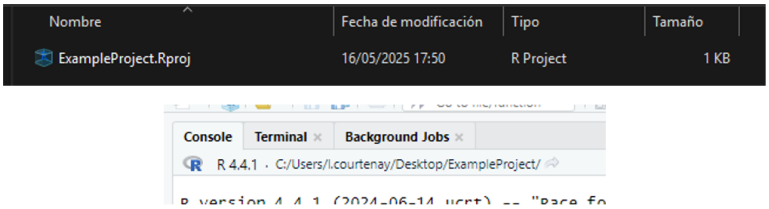

Figure 2. Example of an .Rproj file, and a close up image of the route of where the project is, indicating that R has access to all files in this folder

To open a file where we can start writing code, we need access to the 4th window that is still currently hidden. To open up a new R file, called a script, we go to the tool bar, and on the far left we will find a little dropdown menu that allows us to open a new **R Script**. If we already have a script saved (a file with extension **.R**), then all we have to do is go to the button next to this which lets us search for a file to open, followed by the buttons to save our scripts. The moment we open a new script, the 4th window will appear where we can begin to edit and type R code. Within this window we can define the code we want to communicate with the computer, that will eventually be passed into the console and run. To run a line of code, we can simply go to the top right of this window and find the **Run** button, or simply highlight the line of code we are interested in an type **CTRL+ENTER**. Another option is to run an entire script at once. This is done by clicking on the **Source** button.

Working with R in RStudio

R code is read from left to right, top to bottom. Unlike some other programming languages, it is not necessary to mark the end of a line with a semicolon (as is the case with JavaScript, C++, etc.). However, if we want to, we can use semicolons to write multiple lines of code on a single line. For example:

```r
print("Hello") ; print("People")
print("!")
```

would result in;

```
>> [1] "Hello"
>> [1] "People"
>> [1] "!"
```

Here it is important to talk about "indentations" in programming. An indentation is a tabulation (tab) that separates the line of code from the margin. In most programming languages indents are typically interpreted as the equivalent of two spaces, however in others this may vary and is not so strict. Indentations are a fundamental component of clean code, as it helps us read and interpret the code more easily. While indentation in R is not strictly required, it can be very helpful. As we write code, RStudio automatically adds indentation. But if we want to clean up or adjust the indentation manually, we can use the option in the menu: Code » Reindent Lines (or use the shortcut **Ctrl + I**).

An example of no indentations (and very untidy code that might even potentially produce an error), would be the following;

```r
myfunction <- function (x, y) {
if (y > 0) {
answer = x / y
}
return(answer)
}
```

while a good example, and good practice, would be to write the same as this;

```r
myfunction <- function (x, y) {
    if (y > 0) {
        answer = x / y
    }
    return(answer)
}
```

While indentations ensure that code will run, they also help us see which parts of code belong to which *scope*, which is indicated in R as well by the present of curly brackets, {}, also known as a *code block*. Multiple code blocks should be written with progressively more indentations to make them stand out from the rest of the code, and easily identify what code belongs to which part of the code or function.

Like many text editors, RStudio facilitates the reading of code by colour coding the syntaxis, while it can also provide us with prompts to suggest the code that we are reading, and also flag any potential warnings such as missing commas, or unmatched opening brackets.

An additional tool available in RStudio is its code navigation features. For example, when we type the hashtag symbol (**#**) followed by multiple dashes (**-**), equals signs (**=**), or more hashtags (**#**), RStudio recognizes that we're creating a new section in the code. We can also include the name of the section right after the **#** and before the symbols. This section will then appear in the section index, which we can access from the button at the top right of the editor. The same index can also be found in the bottom-left corner of the interface.

When we define functions in the code, they will also appear in this index, making it easier to navigate between different parts of the script.

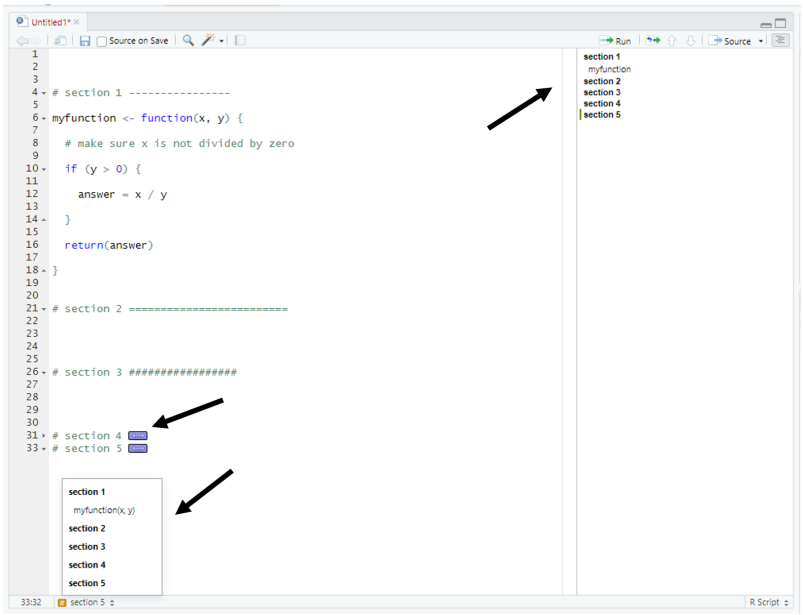

Figure 3. Example of how RStudio allows the use of hashtags, hyphens and other symbols to structure and navegate code in a more efficient manner.

If we look at the left margin of the editor (next to the line numbers), we'll see small arrows next to certain lines. These arrows let us collapse or expand blocks of code. When a function or section is hidden, a blue or purple rectangle will appear, indicating that the content is collapsed. The huge advantage of this is more apparent when we begin working with hundreds to thousands of lines of code, where seeing so much code can be (A) tiring, but also (B) very difficult to navegate.

One of the other major advantages of working in RStudio is the ability to monitor the environment in real time. In the **Environment** pane (top right), we can see that as we create objects, functions, tables, or values, RStudio displays all the elements currently loaded in memory. If we want to inspect the contents of any of these objects, we can either:

Click directly on the object name in the environment, or type `View()` in the console. The built-in viewer in RStudio also renders plots and figures in real time. In the toolbar of this pane, we'll find buttons to browse the plot history, zoom in on plots, as well as export figures in various formats, including **.pdf**, **.png**, **.jpg**, **.tiff**, **.bmp**, **.wmf**, **.svg**, or **.eps**.

Handy Tip

The best quality figures for publications are produced by first saving the figure in `.svg` format, which is vector-based. These files can then be opened in graphic editing software such as *Adobe Illustrator* or *Inkscape*, where they can be tweaked and exported to high-resolution formats like `.png`. This workflow ensures crisp, publication-quality visuals!

Finally, the **Help** tab in the viewer is a powerful tool to consult the official documentation of R and its libraries. For example, if we want to view the documentation for the `plot()` function, we simply run `help("plot")` in the console, and the information will automatically appear in this tab.

Working with libraries in R

Although it is beyond the scope of this course to cover the various libraries available in R, it is inevitable that we mention some of them. A library or package is a collection of functions designed to carry out specific tasks. For example, while R is fully capable of reading and handling tables on its own, there is a large collection of libraries (e.g., tibble) that assist with database management. Likewise, although R offers many tools for creating figures, libraries like ggplot2 can be very helpful for advanced figure design.

In general, R libraries are stored on CRAN (the Comprehensive R Archive Network), although some libraries can be installed through GitHub. To download and install libraries, we can use the `install.packages()` function (as shown in the section titled "Installing R"). The first argument that `install.packages()` requires is the name of the library we want to download (e.g., `install.packages("ggplot2")`). The first time we run this function, R will likely ask us to select a CRAN mirror (server). In that case, we simply choose the server closest to our location. Once libraries are installed, we can load them into memory using the `library()` function, e.g., `library(ggplot2)`. If we want to see which libraries are

currently loaded in our session, we can run the command `sessionInfo()`.

For those interested in installing libraries from GitHub, we can do this by using the devtools library (which requires RTools, see "Installing R" for details). The devtools library provides a function called `install_github()` that can be used to install libraries from GitHub.

Basic R Syntax

As with any programming language, **R** has a set of *lexical rules* that are fundamental for its use. While many of these rules are not strictly required, they are highly recommended to ensure the readability of your code—both for yourself and for other researchers who may work with it.

R is **case-sensitive**, meaning it differentiates between uppercase and lowercase letters. If we return to a previous example, the line `view(table)` is not the same as `View(table)`, `View(Table)`, or `VIEW(TABLE)`. If we want to visualize the table named "table", we need to remember that we defined "table" in lowercase. Also, the function `View()` starts with a capital V. Therefore, the correct code to visualize the table "table" would be: `View(table)`. If we had instead named the table "TaBlE", then the correct line would be `View(taBlE)`.

Additionally, although R can read accented characters (ñ, é, è, ö, ō, β etc.), it is not recommended to use these types of characters. It is best to use characters typically found in English.

Second, like other programming languages, variable, function, or object names cannot contain spaces. For example `my variable` would result in an error, therefore it is better to write `my_variable`.

Handy Tip

There are a large number of other symbols that are highly recommended to either avoid, or are directly forbidden to use. It is good practice to get used to these rules even when talking about things such as file or folder names as well. This includes accented and special characters, as defined before, but also should avoid the use of dots (.), hyphens (-), forward (/) or backward (\) slashes, or overuse of symbols such as commas (,), exclamation marks (!), question marks (?), asterisks (*), hashtags (#), dollar signs ($), ampersands (&), percentage signs (%), the symbol, or anything along these lines.

Comments

A comment in any programming language is a portion of code that is hidden from the computer. We can use comments to annotate parts of our code, write instructions or notes about its use, or, as we saw in previous sections, to mark subsections within RStudio.

Different programming languages use different symbols to indicate a comment. Here are some common examples:

Language	Comment Symbol(s)
R, Python, Julia	`#`
JavaScript, Java, C++	`//`
CSS	`/* */`
HTML	`<!- ->`
LaTeX	`%`

Handy Tip

Remember the rules of clean code, overuse of comments probably means that your code isn't very clear to begin with! It's better to write transparent code that is understandable, than fill your code with comments trying to explain what you have done.

Types of Variables

In programming, a *variable* is a symbolic name that represents a value stored in memory, which can change during the execution of a program. Understanding the different types of variables is essential, as it determines how data is stored, processed, and manipulated. This is also normally where things begin to go wrong, errors occur, and things can get quite tricky.

Literals are values or data that we have not assigned to a variable. Here, we have called them "Types of Values" to simplify the definition. In R, we have the following types of values;

```r
# Numbers

1 # The number 1
1.2 # The decimal number 1.2
5e-2 # The number 0.02 written in scientific format (5 times by 10 to the power of negative 2)

# text

"H" # a character
"Hello" # a string, i.e.  a string of characters

# booleans

T
TRUE
F
FALSE
```

Compared to other programming languages (e.g. C++), R is not strict when it comes to defining numeric values. At its core, numeric values are stored in memory as a double type number. Therefore, to define a numeric value, it is enough to just write it in the console, and R will handle the memory management by itself. As we have seen, R also accepts numeric values in scientific notation, e.g., 5e-2 = 0.02 (something we will see a lot in statistics).

Text in R is mainly defined by using double quotes " ". R also accepts the use of single quotes ' ' to mark text, although it is generally recommended to work with double quotes ("). A character type value usually refers to a single letter, while a set of letters is a string (a character chain).

Boolean values are representations of binary values: true (1) or false (0). It is important to note here that Boolean values are always written in English and in all uppercase letters. R also allows the use of T and F if we want to save space/time.

Handy Tip

Variables such as T and F should be avoided because they can often be considered ambiguous, while in many mathematical formulas the symbol t and f can appear, and lead some programmers to mistakingly overwrite what T and F stand for. This is a potentially huge problem so should be avoided!

Each of these types of values have their own particular properties, and one of the most common causes of errors in R comes from their misuse. In general, R is able to automatically detect the type of value we are using. However, many times problems arise and R can make mistakes. Therefore, it is important to know how to find out, change, or define the format of values.

The function `str()` is used to check the format of a value, while `class()` can also inform us about what something is.

To be able to change or specify the format of something, we use functions of the type `as ... ()` where the dots refer to the type of value we want to use. For example, to store numeric values we can use `as.numeric()`. If we write `as.integer()`, we are forcing R to read the value in integer format (whole number without decimal points). To read and generate text, we use `as.character()`, although it is still mandatory to use quotes if the input is text. For Boolean values, we use `as.logical()`.

Here, we can take advantage of these functions to fix or clean parts of the code. If we want R to read a Boolean value as text, we can write `as.character(TRUE)`. This command would result in `"TRUE"` instead of `TRUE`. On the other hand, if we make a mistake when typing and forget to write TRUE in uppercase, we can use `as.logical("true")` to correct this error.

In R, there are also other types of values or variables called factor variables (or simply factors). Factors are very important in statistics because they allow us to record instances of categorical events, i.e. to represent qualitative variables. A factor is defined using the function `as.factor()`. Factors have additional features and complexities that are worth mentioning, as they are often the source of many issues and misunderstandings. The concept of a "factor" is relatively unique to R and is not found in many other programming languages. For example, although R is implemented in languages like Fortran and C++, neither of these languages have a factor data type. In reality, when R defines a factor, it simply encodes the variable numerically. For instance, if we have a factor with three levels: "low", "middle", and "high", these will be internally represented as 1, 2, and 3. Similarly, levels like "orange", "pink", and "violet" are also encoded numerically. Because of this, factors often require the specification of the ordering of levels — in some cases, the levels can be ranked (ordinal factors), while in others they cannot (nominal factors). We can test this by simply passing a group of factors into `as.numeric()` which would result in turning the supposed qualitative variables we call factors, into a series of numeric values that can often be ordered arbitrarily. *

*This is why some authors have criticised the use of factor levels being blindly passed into functions and algorithms from Machine Learning, where things like order and magnitude often matter.

To ensure that this point is as clear as possible, let's look at an example.

```
factor(c("low", "middle", "high"))
```

Results in;

```
>> [1] low    middle   high
Levels: high low middle
```

Please note that R has automatically ordered the levels of these factors itself, however, has positioned "high" at the lowest level (i.e. numerically this would be a 1), "low" as the second highest level (i.e. 2) and "middle" as the highest level (i.e. 3). This evidently to us does not make much sense. We can prove the earlier described concept here by;

```
as.numeric(factor(c("low", "middle", "high")))
```

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

If we wish to clearly define what the levels are and how they are ordered, we have to provide the following;

```
factor(
    c("low", "middle", "high"),
    levels = c("low", "middle", "high"),
    ordered = TRUE
)
```

```
>> [1] low    middle   high
Levels: low middle high
```

Basic Data Structures

You may have noticed throughout some of the previous examples the use of a peculiar bit of syntax; `c()`. This is our first look at more complex means of storing variables, that fall under the umbrella of vectors, matrices and arrays.

R has the capability to work with vectors, matrices and arrays, as well as changing the format of our data using functions like `matrix()` or `as.matrix()`. Below we will see each of the data types and how they work.

Vectors

A vector can be defined using the `c()` function, which stands for "combine". For example:

```
c(1, 2, 3, 4, 5)
```

This creates a numeric vector. We can also combine strings, qualitative variables, or any other type of variable:

```
c("Hello", "World")
```

Note, however, that if you are using multiple types of variables, R will force all the variables to be the same type, which can be problematic. If you introduce `c(1, "hello", TRUE)` into the console, R will force all of the values to be strings, while `c(1, TRUE)` will result in two integers, 1 and 1.

Although the function `as.vector()` exists and can coerce other data types to vectors, it is rarely necessary when creating vectors directly with `c()`.

Matrices

To define a matrix in R, we use the `matrix()` function. For example:

```
matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3)
```

This creates a 2-row, 3-column matrix. The values are filled in column-wise by default. If we use `as.matrix()` on a vector:

```
matrix(c(1, 2, 3, 4, 5, 6))
```

we obtain a 5×1 column matrix. To control the shape and structure of a matrix, we should use the `nrow` and `ncol` arguments directly in `matrix()`.

Arrays

Arrays are closely related to both of the previously defined data structures, however they have an additional complexity in dimensionality. A 1-dimensional array is literally the same as a vector. A 2-dimensional array is synonymous to a matrix. A 3-dimensional array would be similar to if we were to stack two matrices on top of each other. In linear algebra and mathematical literature, these are often referred to as tensors as well. A good example of this is an image; an image is represented numerically

by the computer as 3 matrices stacked on top of each other containing light intensity values for Red (R), Green (G) and Blue (B) channels. This would be a 3-dimensional array. In geometric morphometrics, we unknowingly use arrays all the time, the `geomorph` library, for example, stores landmarks as 3-dimensional arrays, where the first dimension is the number of landmarks, the second dimension are the x, y and sometimes z coordinates for each landmark, and the third dimension is each of the individuals being studied.

An array, therefore, is a further generalisation and can have any number of dimensions. However, it is more common and better practice to use the terms vector and matrix for lower dimensional arrays.

Arrays can be created using the `array()` function. The difference here is the introduction of the argument `dim` (i.e. dimension). `dim` defines the dimensionality of the array, in the form of a vector. Consider the following examples:

```
array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12))
```

```
>> [1] 1 2 3 4 5 6 7 8 9 10 11 12
```

```
array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim = c(2, 6))
```

```
>>      [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
   [1,]    1     3     5     7     9    11
   [2,]    2     4     6     8    10    12
```

```
array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim = c(6, 2))
```

```
>>      [,1]  [,2]
   [1,]    1     7
   [2,]    2     8
   [3,]    3     9
   [4,]    4    10
   [5,]    5    11
   [6,]    6    12
```

```
array(c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim = c(3, 3, 2))
```

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

, , 2
      [,1]  [,2]  [,3]
   [1,]   10     1     4
   [2,]   11     2     5
   [3,]   12     3     6
```

Warning

The order in which R fills in the matrix or array can often be quite confusing, and we need to be careful that this is done in a specific way. In the `matrix()` function the parameter `byrow = TRUE` can be used to ensure that R fills the matrix across the rows first, but by default this parameter is set to `FALSE`. For arrays, R always fills in the data in column-major order, starting with the first dimension, then the second and so forth. There is no `byrow` parameter for `array()`, so if you want a very specific order, you have to be careful with the way in which the data is provided. Alternatively, as we will see later on, it is often better to create an empty array, and fill it in afterwards, to ensure the ordering aligns with what we are looking for.

Handy Tip

Don't forget, that when in doubt the R documentation is the best source of information to look at! Try typing `help(matrix)` or `help(array)` and see what the official documentation says.

The `:` operator and sequences

One of the simplest and most commonly used operators in R is the colon operator (`:`). This operator is used to generate a sequence of numbers between two values. In plain terms, it answers the question: "What are all the whole numbers from this number to that number?". For example:

```
1:5
```

```
>> [1]  1 2 3 4 5
```

While R also automatically creates the sequence from the first number to the second in steps of 1. For example, using `5:1` would produce the opposite:

```
>> [1]  5 4 3 2 1
```

This is especially useful when we need to repeat something over a range of values, such as when we use loops or index vectors and dataset in later sections of this document.

Declaring variables and storing them in memory

In R, a variable is defined using either the equals sign `=` or the assignment arrow `<-`. According to R's documentation, it is recommended to use `<-` for variable assignment and `=` for function arguments. However, for users familiar with other programming languages such as C++, JavaScript, Java, or Python, it is also acceptable to use `=` exclusively.

To assign a new variable, we write the variable name, followed by `<-`, and then the value to be assigned. For example:

```
individual <- 5
name <- "Homo erectus"
height = 150.5
bipedal <- TRUE
```

As in other programming languages, variable names in R cannot start with a number. Furthermore, unlike languages such as JavaScript or Python, R does not allow variable names to begin with special characters like `_` or `$`. Thus, variable names must begin with a letter.

To inspect the contents of a variable, either:

- Check the **Environment** tab in RStudio, or
- Type the variable's name in the console.

Handy Tip

Remember R is case sensitive, if the variable has been declared as `x`, then this variable will always exist with the lower case letter x as its name. For this reason declaring both `x <- 1` and `X <- 2` will work, but is highly problematic and not at all a good example of clean code.

Handy Tip

Variable names should be informative to what they store, naming something `x` is not good practice unless it is meaningful to what `x` is, for example x from a formula, $f(x) = y$, or the x axis of a set of coordinates. This should be clear, though, to the user.

Mathematical operations

There are large number of mathematical functions and operators available. While many are similar across all programming languages, some are very particular to R. The following table presents some of the operators;

Description	Operator	Formula	Example	Output
Addition	`+`	$2+3$	`2 + 3`	5
Subtraction [b]	`-`	$2-3$	`2 - 3`	-1
Multiplication	`*`	2×3	`2 * 3`	6
Division	`/`	$2\div 3$	`2 / 3`	0.67 [a]
Power	`^` or `**`	2^3	`2^3` or `2**3`	8
Dot Product	`%*%`	$2\cdot 3$	`2 %*% 3`	6
Modulus (remainder)	`%%`	$2 \bmod 3$	`2 %% 3`	2
Integer Division	`%/%`	$\lfloor 2/3 \rfloor$	`2 %/% 3`	0
Negation [b]	`-`	-2	`-2`	-2

[a] Approximate decimal value shown for clarity.
[b] Negation and subtraction use the same symbol, but their function differs depending on context.

Don't worry if you are not familiar with what all of these things are, the purpose of this table is to provide a reference for if you come across them in the future.

For mathematical functions, the following table provides some examples of different operations we can perform in R

Function	Description	Example	Output
`abs(x)`	Absolute value of `x`	`abs(-5)`	5
`sqrt(x)`	Square root of `x`	`sqrt(16)`	4
`log(x)`	Natural logarithm (ln) of `x`	`log(10)`	2.302585
`log10(x)`	Base-10 logarithm of `x`	`log10(100)`	2
`log2(x)`	Base-2 logarithm of `x`	`log2(8)`	3
`exp(x)`	Exponential (e^x)	`exp(1)`	2.718282
`round(x, n)`	Round `x` to `n` decimals	`round(3.14159, 2)`	3.14
`ceiling(x)`	Round `x` up	`ceiling(2.3)`	3
`floor(x)`	Round `x` down	`floor(2.7)`	2
`trunc(x)`	Truncate decimals from `x`	`trunc(-2.7)`	-2
`sin(x)`	Sine of `x` (radians)	`sin(3.14159 / 2)`	1
`cos(x)`	Cosine of `x` (radians)	`cos(0)`	1
`tan(x)`	Tangent of `x` (radians)	`tan(3.14159 / 4)`	1
`sum(x)`	Sum of elements in `x`	`sum(c(1,2,3))`	6
`prod(x)`	Product of elements in `x`	`prod(c(2,3,4))`	24
`mean(x)`	Arithmetic mean	`mean(c(4,6,8))`	6
`median(x)`	Median	`median(c(4,6,8))`	6
`mad(x)`	(Normalised) Median Absolute Deviation	`mad(c(4,6,8))`	2.9652
`sd(x)`	Standard deviation	`sd(c(1,2,3))`	1
`var(x)`	Variance	`var(c(1,2,3))`	1
`min(x)`	Minimum value in `x`	`min(c(5,3,9))`	3
`max(x)`	Maximum value in `x`	`max(c(5,3,9))`	9

Table 1. Common Mathematical Functions in R

A detailed explanation and exploration of a more programmatic approach to mathematics is provided at the end of this document. However, for the purpose of understanding syntax, this should be enough for the moment.

Handy Tip

In cases where R does not have certain functions that we might be looking for, there is always the possibility to find many of them in external functions from specific libraries, or even more fun (for some), to program them yourselves.

Finally, there are some (although not many) predefined mathematical constants directly available in R. The main one is π, which is simply defined as `pi` in R. Another potential example is Euler's constant, e, which is available through the `exp()` function. To get Euler's constant, you can type `exp(1)`, however the `function` is more typically used to calculate things like e^x; `exp(x)`.

Handy Tip

It is important to mention that all of these operators can be applied to things like vectors and matrices in a similar fashion. If you define a vector `x <- c(1, 2, 3)` then `x` can be multiplied by another value, say 2, as follows; `x * 2`, resulting in 2, 4 and 6. This begins to become fundamental when considering how to perform specific matrix operations that are common in linear algebra an calculus.

Logical Operators (`&`, `|`, `!`, `=`, `<` and `>`)

Logical operators are what allow us to compare two values, such as calculating whether a given variable is higher or lower to another variable, equal to, or not equal to. These are incredibly useful operators to know and understand and can drastically improve code if handled correctly.

If we take two variables;

```
first_value <- 10
second_value <- 20
```

We can compare them by using varying combinations of the operators `<`, `>`, `=`, `!`, `&` and `|`. The first two of these operators are relatively self explanatory; is one value higher or lower than another. If we use these operators, R returns a boolean value informing us of whether the condition is met or not;

```
first_value < second_value
```

```
>> [1] TRUE
```

```
second_value < first_value
```

```
>> [1] FALSE
```

```
second_value > first_value
```

```
>> [1] TRUE
```

```
first_value > second_value
```

```
>> [1] FALSE
```

If we would like to know whether one value is greater than, or equal to, another (e.g. $x_1 \geq x_2$), we can combine the `>` operator with the `=` operator as follows;

```
first_value >= second_value
```

```
>> [1] FALSE
```

The same can be said for $x_1 \leq x_2$ by simply changing `>` for `<`. In this case it is important to be careful with the `=` operator, because this operator by itself is used to store the value of something in a variable. For this reason, if we want to see if two values are equal, we use a combination of two `=` operators

```
first_value == second_value
```

```
>> [1] FALSE
```

Warning

Just to clarify, `first_value = second_value` would result in replacing the content of `first_value` with that of `second_value`, resulting in both variables now containing the exact same data. It would be the same as if we were to type `first_value <- second_value`. The correct way of seeing whether two things are equal or not is to use `first_value == second_value`.

The `!` operator is the next fundamental operator we must learn, which is called the logical negation operator. This operator negates something, for example, `!=` would be a way of testing if two things are NOT the same. Likewise, using `!` next to something that is `TRUE` would make it `FALSE`, and vice versa.

Logical operators `&`, `&&`, `|`, and `||` allow us to combine multiple logical expressions. They are used to perform logical AND and OR operations, but with important differences in how they evaluate expressions. For example, if we want to check two conditions, we can use these operators to combine multiple comparisons; `1 < 2` and `3 > 2` could be combined using `1 < 2 & 3 > 2`, which would result in `TRUE`.

Handy Tip

It's important to avoid ambiguity and can often be much cleaner to use parentheses in these types of comparisons. So for example `1 < 2 & 3 > 2` would be better read as `(1 < 2) & (3 > 2)`.

The operators `&` and `|` work element-wise when applied to vectors. This means that each corresponding element of two logical vectors is compared, and a logical vector of the same length is returned. The operators `&&` and `||` perform short-circuit evaluation and only check the first element of each vector, which can be useful for very specific applications.

While these operators may seem very abstract, they will become very important in later more complex components of programming. It is therefore important to simply understand what each symbol means, and how they can be combined to ask a series of questions.

Special Values in R

In R there are several special values that represent relatively abstract concepts like infinity, undefined numbers, missing values, or empty objects. These are part of the language and behave in specific ways, and can often be very confusing or frustrating if you have never encountered them before.

`Inf` and `-Inf`

`Inf` and `-Inf` literally represent the concept of positive and negative infinity. This is a very abstract concept, and can often be frustrating to encounter because you cannot perform many operations on them, and they pop up in quite a lot of different contexts unexpectedly. This stems, in part, from a number of mathematical theoretical ideas that can be tricky to understand.

One of the most frequent means of coming across `Inf` and `-Inf` is if you divide by zero. In mathematics, division by zero is referred to as an undefined concept. What does this mean? Well... literally that, it is undefined.

Division basically asks the question "How many times does this number fit into that number?". For example, $6 \div 2 = 3$, because 2 fits into 6 exactly 3 times. In other words, in mathematics if you divide a given number by another number, the result can then be multiplied by the denominator to recover the numerator. For example, $4 \div 2 = 2$, so $2 \times 2 = 4$, or $21 \div 3 = 7$ so $7 \times 3 = 21$.

So, going back to the previous example of $6 \div 2 = 3$, if we were to replace 2 with 0 and ask the question, how many times does 0 fit into 6, then the answer does not make any sense, as no number multiplied by 0 will ever equal 6; $6 \times 0 = 0$. Mathematically, division by zero does not have any meaning, this is why it is typically called "undefined".

R has a very strange way of dealing with this concept, which is unique to many other programming languages. R handles the division by zero in a way that makes calculations practical, even if they are not necessarily mathematically rigorous. Mathematically, as the denominator of a fraction gets closer and closer to zero from the positive side, the fraction $1/x$ will approximate infinity; it grows without bound exponentially. This is why R uses infinity to describe this. If you compute `5 * Inf`, it will return `Inf` as opposed to an error, that means you can still run your code without getting errors. Nevertheless, in many cases this is the source of many problems and likely means we have done something wrong.

Key Takeaway

In summary, if you see the value `Inf`, it likely means you have a mistake somewhere where one of your calculations tries to divide something by 0. This is something we should always avoid, as I really doubt any archaeologists are working with such complex mathematics that a concept of infinity is needed. This is therefore a good moment to go back and check your calculations, because this is not something we can do in mathematics!

`NaN`

`NaN` stands for "Not a Number", and is returned when an operation is undefined. While we spoke before about how technically any number divided by 0 is undefined, R deals with other undefined theories as NaN instead. A good example is `0 / 0`. This value is used specifically for invalid numerical results. We can test whether something is NaN by using the function `is.nan()`, while `!is.nan()` would negate this.

```
is.nan(0/0)
```

```
>> [1] TRUE
```

```
is.nan(1*0)
```

```
>> [1] FALSE
```

NA

NA is one of the most frustrating, yet common occurances in R code and is normally when something has gone wrong. NA stands for "Not Available", and is R's way of handling missing or undefined data. We can check if something is NA by using `is.na()`, and negate this using `!is.na()`. Performing any calculation on NA will result in another NA, which is why its so frustrating. If you get an NA value as output of a function, it simply means that at some point R has encountered another NA value and this error or mistake has propagated throughout the code.

```
c(1, 2, 3) * c(2, NA, 5)
```

```
>> [1] 2 NA 15
```

```
sum(c(1, 2, 3) * c(2, NA, 5))
```

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

Some functions offer a way of getting around the presence of NAs in our data by including a parameter called `na.rm`. For example, in the function `mean`, which calculates the arithmetic mean of a set of values, we can ask R to calculate the mean without considering these NA values;

```
x <- c(1, 2, NA, 4)
mean(x)
```

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

```
x <- c(1, 2, NA, 4)
mean(x, na.rm = TRUE)
```

```
>> [1] 2.333
```

Handy Tip

Just because R offers the option of not taking into account potential NA values, does not mean we should abuse it. `na.rm` can be incredibly problematic if overused, and is not recommended. It is always best to check why the NA appears, than ignore it. Use `is.na()` to try and find where the NAs occur, and see if the problem can be fixed, or remove those observations from the mix.

NULL

NULL, unlike NA, NaN and `Inf`, can be incredibly valuable in programming if used right. As opposed to NA, where NA refers to data that is missing, NULL simply means that the object or value does not exist at all, it is simply empty. If we declare `x <- NULL`, then it means that `x` is now declared, but it is empty. This can be potentially useful because it means that R already knows that `x` kind of exists... but not quite yet, and we can later use this to fill it in later;

```
x <- NULL
print(x)
x <- 1
print(x)
```

```
>> [1] NULL
>> [1] 1
```

We can check if something is NULL by using `is.null()`.

Information

NULL and NA are often confused, but to clarify that NULL is simply an empty object, you can as how long a vector is using the `length()` function, so if you type `x <- NULL` followed by `length(x)`, you will get the value 0 (the vector is empty). However, if you do the same with NA, the length will be 1, because the entry exists but is missing. These are two different things.

Special Operators in R

There are a series of extra operators that are worth mentioning, as they can be incredibly useful, although we may not see them as frequently if we are not advanced users of R. Here we will outline what the `::`, `%in%` and `|>` operators are, and how we can leverage them for more efficient programming.

There are also a fair few non-standard operators that are not also relatively common, but can only be seen in specific libraries and that are worth mentioning. Here we will outline the `+` symbol in ggplot2, which is slightly different to the standard `+` mathematical operator, as well as the pipeline operator `%<%`, which is common in many libraries.

The `::` operator

The `::` operator is an incredibly important operator in many programming languages, including R. Syntactically, this comes in part from languages such as C and C++ where the concept of a **namespace** is fundamental.

A *namespace* is a way of organising code so that different packages or modules can define their own functions and objects - even if they have the same names - without causing confusion or conflicts. In R, a *namespace* is the same as an R library.

The `::` operator allows us to access a specific component of a specific R library. For example, many libraries may have functions or objects with the exact same name. While functions and objects will be explained later in this document, it was decided to introduce the idea of the `::` operator here so as to avoid confusion with the already defined `:` operator that is used to define sequences of numbers.

To use the `::` operator, all we have to do is provide the name of the namespace, in this case the R library, followed by the `::` operator, and finally the name of the object or function we wish to get access to. For example, many R libraries might have functions called `plot`, and this might create confusion and errors when they are all loaded at the same time. If we wish to access the `plot` function from a specific library, all we have to do is type `package_name::plot()` and we can access the `plot` function from that specific library. Likewise, if we want to access a specific function from *base R*, i.e. the original functions already defined by R, we can type `base::plot()`.

Likewise, if we are writing our own functions in our script, and we happen to write a function with the same name as one that already exists (effectively overwriting the original function), we can get back the functionality of the original code by calling it from its given namespace. This will become clearer when we get to the part of the document specifically about functions.

> **Handy Tip**
>
> The fact that multiple libraries may share functions with all the same names is often (but not always) an example of unclean code. If some libraries were to write functions with clearer names, we might not have the need to create so many functions with the same name. This is worth keeping in mind when making your own functions!

The easiest way to see (A) what a namespace is, and (B) what namespace a function belongs to is to type the name of a function, but without the parentheses at the end. In other words, if you type `plot` as opposed to `plot()` the following will appear on the screen;

```
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x0000016440bd2460>
<environment: namespace:base>
```

The important part here is to see the argument `<environment: namespace:base>` that indicates to us where the function `plot()` is stored.

The `%in%` operator

The `%in%` operator is a fun operator that can be really useful for certain tasks such as querying a database. The `%in%` operator is a logical operator, much like `<`, `>`, `==`, `!`, `&` and `|` that were described previously. `%in%` tests whether one element exists inside of another one, for example we can scan through a vector of values to see if a given value is present by using `%in%`. This can be incredibly useful for things such as trying to check if a value or set of values belong to a list of possible acceptable options. For example;

```
human_species <- c("H_erectus", "H_neanderthalensis", "H_sapiens")
"A_afarensis" %in% human_species
```

```
>> [1] FALSE
```

Likewise, we can check for multiple values through;

```
c("A_afarensis", "H_ergaster") %in% human_species
```

```
>> [1] FALSE  TRUE
```

Unfortunately, there is no native R operator that does the opposite of `%in%`, i.e. there is no way to negate this operator already built into R. However, I came up a way to do this using the following;

```
`%!in%` = Negate(`%in%`)
```

which can easily be used by;

```
c("A_afarensis", "H_ergaster") %!in% human_species
```

```
>> [1] TRUE  FALSE
```

The `|>` operator

`|>` is the pipe operator, a special operator used in R to pass the result of one expression as input into the next. This allows code to be cleaner and more readable in instances where we need to chain operations together.

Instead of writing long lines of code like this:

```
round(mean(c(1, 2, 3, 4, 5)), 2)
```

```
>> [1] 3
```

Where we are rounding the mean of a vector. What we can do instead is first define the vector, then pass it into the `mean()` function, and finally into the `round()` function, like this;

```
c(1, 2, 3, 4, 5) |> mean() |> round(2)
```

```
>> [1] 3
```

which many people find easier to read, and encourages clearer logic to handling data.

The `%<%` operator from magrittr

The `%<%` operator is also a pipe operator, and is thus an alternative to `|>`, however is not native to R. In fact, `%<%` actually predates `|>`, hence the eventual implementation of such an operator into the native R syntax. `%<%` is the pipe operator from the **magrittr** package, and is implemented in many, many other libraries, such as those that form part of the **tidyverse** family such as **tibble** and **dplyr**. The functionality is in fact identical, but allows more flexibility for certain functions, and is still the most popular implementation of such an operator.

Information

The base R pipe `|>` was introduced in R 4.1.0. If you're using an older version of R, it may not be available. In that case, you can still use `%>%` from the `magrittr` or `dplyr` packages.

The `+` operator from ggplot2

The `+` is a very simple operator from the **ggplot2** package, but warrants mentioning simply to help define how it is different from the `+` operator already defined in the section about Mathematical operations. **ggplot2** is a library for data visualisation, and the creation of highly personalised and visually attractive plots and figures. **ggplot2** works by establishing the figure and plot as a series of layers where the `+` operator helps stack the layers of the visualisation on top of eachother. For example, if you take a base plot and want to add a personalised x-axis label, you can use `+` to add another layer on top of the plot so that the final plot is more customisable than what is achieved in the default visualisation.

Complex Data Structures

We have already seen in previous sections what vectors, matrices and arrays are. Nevertheless, there are slightly more complex data structures that are incredibly useful and common in R that we must cover before we can go onto things like functions, objects and other more advanced components of the R programming language. These are lists and data frames; the core of handling data and databases in R.

Information

A word that will appear a lot throughout this section is **object**. While we have not yet defined what an object actually is in R, or in programming more broadly, it is not essential to understand the full technical details in order to start working with things like lists and data frames. Objects will be explained more thoroughly later in this document.
For now, it's enough to know that an **object** is just something that stores a value — such as a number, a piece of text, a table, or a more complex structure like a model or a list — and that we give it a name so we can refer back to it later.

Lists

In a previous section we saw how there are many types of variables, including characters, strings, booleans, numbers, etc. We also saw in basic datastructures that we can store a collection of these variables in a vector, using the `c()` function. However, we also saw how vectors cannot store variables of different types, such that the following;

```
c(1, "H_erectus", TRUE)
```

will result in R converting all of the values to strings;

```
>> [1] "1"  "H_erectus"  "TRUE"
```

A list in R is a data structure that allows us to store multiple elements of different types into a single object, overcoming this issue and limitation. The magic of lists is that they can even go as far as to store lists within them, as long as the memory of our computer allows us to do so.

Information

Lists can hold pretty much anything, including numbers, characters, strings, booleans, vectors, matrices, arrays, data frames, other lists, functions and models. They are versatile and can do pretty much anything. They are a fundamental component of most R objects!

Lists can be created very easily using the `list()` function. If were to redefine the above vector using `list()` instead,

```
list(1, "H_erectus", TRUE)
```

we would get;

```
>> [[1]]
   [1] 1

   [[2]]
   [1] "H_erectus"

   [[3]]
   [1] TRUE
```

Notice here that the number 1 and the value for `TRUE` are no longer represented as strings, they are now instead stored each with their own corresponding datatype, as we intended. We can also name the elements within this list, which makes code not only easier to read, but much cleaner and easier to work with;

```
list(
    individual = 1,
    species = "H_erectus",
    bipedal = TRUE
)
```

```
>> $individual
   [1] 1

   $species
   [1] "H_erectus"
```

```
$bipedal
[1] TRUE
```

In this case note that instead of each element in the list being labelled from 1 to 3 inside of square brackets `[[]]`, the elements are now named. This is not just informative, but lets us access more efficiently the value of certain elements. This is referred to as **indexing**, and will be described in detail later on.

Handy Tip

Remember how important using meaningful names is for the definition of the names of our variables, it is even more relevant here! When defining a list, it is much better to name each element of the list, otherwise we can very quickly loose track of what we are working with, which leads to mistakes and quite often bad science!

Data Frames

Data frames are another one of the most important types of data structure in R. Data frames are more what we would typically refer to as tables, spreadsheets, or even databases, however their underlying definition is a bit more complicated than that.

Data frames contain columns and rows, much like a spread sheet, where columns typically refer to values of a given variable, while rows contain observations or records. Data frames differ from matrices, for example, in how the columns can contain different types of data, much like a list, however all values in a single column must be the same type and do not accept as many different types as lists. A dataframe is therefore like a list of equal-length vectors, each vector representing a column.

Data frames can be defined with the function `data.frame()`. Once again we can pass vectors into the function like this;

```
data.frame(
    c(1, 2, 3),
    c("H_erectus", "H_sapiens", "C_elaphus"),
    c(TRUE, TRUE, FALSE)
)
```

however it would make much more sense if the columns were named and labeled, like so;

```
data.frame(
    individual = c(1, 2, 3),
    species = c("H_erectus", "H_sapiens", "C_elaphus"),
    bipedal = c(TRUE, TRUE, FALSE)
)
```

While we could have technically defined the same thing using a list, the structure and manipulability of a data frame is unique, and the type of object we use will depend quite a lot on what we are doing. Likewise, data frames can only hold vectors in their columns, while lists can hold many things, therefore the efficiency of using certain objects for certain tasks will also depend on that.

Warning

Data frames can only hold columns and rows of equal length, i.e. the vectors inserted as columns have to all be the same length. If we work with vectors that are not of equal length, a list would be more suitable. This is a very common source of error and is frequently a cause for frustration.

Indexing and using $, [ ] and [[ ]] correctly

Indexing is the process of accessing specific elements within a data structure, such as a vector, matrix, array, list, or data frame. It allows us to extract and filter data efficiently, enabling a wide range of operations and calculations. There are two main symbols we need to know to perform this task in R; the dolar sign, `$`, and the square brackets [and].

Before diving into how each of these works, it's important to understand one key concept: R does not use zero-based indexing, unlike many other programming languages. This means that indexing in R starts at 1, not 0 (the first column of a table in R is the column number 1, however in python it is column number 0!)

This distinction is one of the most common sources of confusion for users transitioning between R and languages like Python, which do use zero-based indexing.

Information

If this is your first time working with programming, or if you have no plans to use other languages such as Python, this detail might not seem immediately important. However, if you ever do work across languages, understanding how indexing conventions differ is incredibly useful and will help you avoid subtle errors.

There are several ways to access elements within an object, but we will start from the ground up by looking at basic data structures such as vectors, before moving on to more complex types like lists.

Indexing a Vector

As previously discussed, vectors are one-dimensional arrays. They are the simplest data structure in R that goes beyond a single scalar value, and can be indexed easily using square brackets `[ ]`, enclosing the index of the value we wish to access.

For example, if we were to type the name of a vector, followed by `[1]`, we would be accessing the first value in the vector. Likewise, if we were to type the name of the vector, followed by `[12]`, we would be accessing the 12th value stored in the vector. Here is an example;

```
my_vector <- c(10, 20, 30, 40, 50)
my_vector[2]
```

```
>> [1] 20
```

Likewise, we can use this to modify a given value inside of a vector, say:

```
my_vector <- c(10, 20, 30, 40, 50)
my_vector[2] <- 55
```

would result in `my_vector` containing the values 10, 55, 30, 40 and 50.

Another thing we can do is filter the vector given a set of conditions. For example, if we only want values of the vector that are under the value 45 we can use our logical operators. The line of code `my_vector < 45` would result in;

```
>> [1] TRUE FALSE TRUE TRUE FALSE
```

so using `my_vector[my_vector < 45]` would result in;

```
>> [1] 10  30  40
```

Information

As you can see we are now beginning to combine many of the building blocks behind programming to get slightly more functional commands. This is truly what understanding all this theory is for! Imagine we now have a vector of sample labels, qualitative attributes or measurements, indexing and filtering using things like logical operators would allow us to do a large number of things!

Handy Tip

We can therefore use literally anything and everything we have already come across to index in more complicated ways. For example, if we wanted to extract the first, third and fifth element of a vector, we could pass `c(1,3,5)` into the square brackets. If we wanted the first 5 elements, followed by the 7th, and then elements 11 to 22, we could use `[c(1:5, 7, 11:22)]`. Be creative and remember all the tools we have seen up until this point!

Indexing a Matrix

As shown in previous sections, a matrix is a 2-dimensional array, containing rows and columns. To index a matrix, just like vectors, we use square brackets. However, unlike a matrix, now we have to consider the dual dimensionality of the matrix, and it is not enough to simply indicate a single number inside these square brackets. The basic recipe for indexing a matrix is;

```
matrix[row, column]
```

The fundamental component here that we have to pay attention to is the comma within the square brackets. The comma is how we identify the specific dimension we are aiming to index. If we wish to access the first column, we would have to place a 1 after the comma. However if we wish to access the first row, we would place the 1 before the comma. If we would like to access the value that lies in the first column of the first row, then we would place a 1 both before and after the comma, as follows;

```
my_matrix <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3, nrow = 3)
print(my_matrix)
```

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

```
my_matrix[,1]
```

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

```
my_matrix[1,]
```

```
>> [1]   1     4     7
```

```
my_matrix[1,1]
```

```
>>  [1]    1
```

```
my_matrix[2,3]
```

```
>>  [1]    8
```

Key Takeaway

The first value in the square brackets indicates the first dimension of the matrix; the rows. This is followed by a comma. The second value indicates the second dimension of the matrix; the columns. If we leave a value blank before or after the comma, it implies that we want all values belonging to that specific indexed row or column.

Indexing an Array

Now that we have seen how to index vectors and matrices, understanding the indexing of arrays is relatively simple, it is an evolution of the previously defined concepts, much like how an array can technically be used to represent both vectors and matrices as well.

```
array[1st dimension, 2nd dimension, ..., nth dimension]
```

The number of commas required between the square brackets simply increases with the number of dimensions the array has, while the indexing works very similar. It might be easier in this case to provide some actual examples, as some people still find the idea of multidimensional arrays to be quite difficult.

In geometric morphometrics, for example, arrays are fundamental. They are the means in which we store our landmark data. The first dimension (`[x, , ]`) is referred to as p in the literature, which is the number of landmarks. The second dimension (`[ ,x, ]`) is reffered to as k, which is the number of dimensions, $k = 2$ for 2-dimensional landmarks, and $k = 3$ for 3-dimensional landmarks. Finally, the third dimension (`[ , ,x]`) is referred to as n, which is the number of individuals. If we want to extract, for example, the x, y and z coordinates of the fifth landmark across all individuals we would use `[5,,]`, and if we wanted this for only the first 5 individuals we would use `[5,,1:5]`. If we wanted to extract all individuals that belong to a specific species, and only take the x coordinates for landmarks 11 to 150, we could use

```
[11:150, 1, species == "H_sapiens"]
```

Another good example of arrays are images. Images typically have three channels, therefore can be represented as 3-dimensional arrays, where the first dimension represents the rows of pixels, the second the columns of pixels, and the third dimension the R, G and B channels.

You might have noticed that beyond three-dimensional arrays, I have not given any examples. This is literally because I am unaware of a single archaeological case study where we would ever need higher dimensional things like advanced tensors. This is something much more related to the fields of physics and advanced mathematics than what we typically see in archaeology.

Indexing a List or Data Frame

Just as how we have been able to see the evolution of how to index values from vectors, through to arrays, data frames naturally follow this evolution, while adopting all of the rules we have already seen up until this point. In a data frame, we can index the columns by using `[,x]`, while the rows are indexed using `[x,]`.

The additional complexity of a data frame however comes from the fact that we can name the columns of our data frames, which introduces the `$` symbol into play, one of the most important components of the R programming language. `$` is used to index a component of an object by name; given that we know the name of something inside an object, we can call that element directly using the `$`. This is just as true for lists, where we can also use `$` to index a part of a list that we know the name of.

Let's take the previous example of a data frame that we defined when introducing what a data frame is;

```
my_example_data <- data.frame(
    individual = c(1, 2, 3),
    species = c("H_erectus", "H_sapiens", "C_elaphus"),
    bipedal = c(TRUE, TRUE, FALSE)
)
```

If we wish to access the first column of this data frame, we could use `my_example_data[,1]`, however, given that we know that the first column is called "individual", we could also use `my_example_data$individual`.

Lists, on the other hand, can get even more complicated, and this is where some people begin to have some confusion, especially if they are not as familiar with these types of data structure.

Think of a list in R as a collection of containers. Each container can hold anything, a number, a string, another list, etc. However, lists are not homogeneous in size like all the other data types. The first element of a list can be shorter than the second, and this would not produce a problem. There are three main ways to index a list. Two of them, we have already seen; the use of a single square bracket, like `my_example_list[1]`, or by using `$`, like `my_example_list$variable_one`. The third approach is by using double square brackets, which we will look into next.

Let's first define an example list;

```
my_example_list <- list(
    individual = c(1, 2, 3),
    species = c("H_erectus", "H_sapiens", "C_elaphus"),
    bipedal = c(TRUE, TRUE, FALSE),
    meta_data = list(
        site = "atlantis",
        date = 3900000,
        excavation_campaigns = c(1919, 2020, 3030)
    )
)
```

As you can see this list contains a bunch of different things, ranging from vectors containing strings, numbers, boolean values, and even another list which contains a string, a number, and a vector. This is a good proof of concept that lists can literally contain anything, even if they contain things that are not all the same length either. If we want to extract the list of species, all we have to do is type `my_example_list$species`. If we wish to access the list within the list called `meta_data`, we can do the same, and we can even extend this further by accessing `excavation_campaigns` using `my_example_list$meta_data$excavation_campaigns`. We can extend this *even further* by combining everything we have seen so far, by getting the first and last excavation campaign using

```
my_example_list$meta_data$excavation_campaigns[c(1, 3)]
```

The options are endless (I hope you see how everything fits together!).

The issue with lists begins to arise when using `[]`. If we were to run the following code;

```
my_example_list[1]
```

we would get

```
>>
   $individual
   [1]  1  2  3
```

Why is this different? Please note the appearance of `$individual` that appears directly above the result we were looking for. This means that `my_example_list[1]` has not actually returned the vector, but in fact a sub-list. Confused? Me too. What indexing a list with just one square bracket does is return another list with only the element we are looking for. I am not 100% sure in what cases this would be useful, but its the reality of how R works, and how we can often get quite a lot of errors if we use the single square brackets with a list.

Another option we have, therefore, is to use double square brackets. If we were to run `my_example_list[[1]]`, this *would* return the vector, and not another list. This is the way in which we can index the first element and return it in the format it was originally stored as. From this we need to highlight that if we wish to extract the contents of a given part of the list, we have to use `$` or `[[]]`, however if we wish to extract something while preserving the list structure, we use `[]`.

Key Takeaway

Indexing in R can either be done using a the `$` sign, or square brackets, depending on whether we know the name of the element we are looking for, or its position inside the object, *starting from 1*. Depending on the dimensionality of the object we are indexing, we have to use an appropriate number of commas within square brackets to indicate not only the position of what we are indexing, but from which dimension. Lists are a slight exception where sometimes using `[]` can lead to a result which might not be 100% what we are looking for. While some people therefore try to avoid using lists, we will see in the section about functions and object oriented programming that lists are unavoidable. It's therefore useful to know what methods to use, and when, to avoid future headaches!

There are some additional complexities and methods for indexing in R, but with combinations of the above rules, you can perform almost every operation necessary to perform most tasks. One thing that I have failed to mention is that the double square brackets *can* be used in other types of data structure, such as data frames, however this is less common. For example, if we know the name of the column of a data frame, we can also pass this name into double square brackets and acheive the same function as if we were to just use the `$` sign. This is slightly messy though, and I suggest sticking to the typical `$` to avoid confusion. Bellow is a figure of all the different ways you can extract information from a data frame.

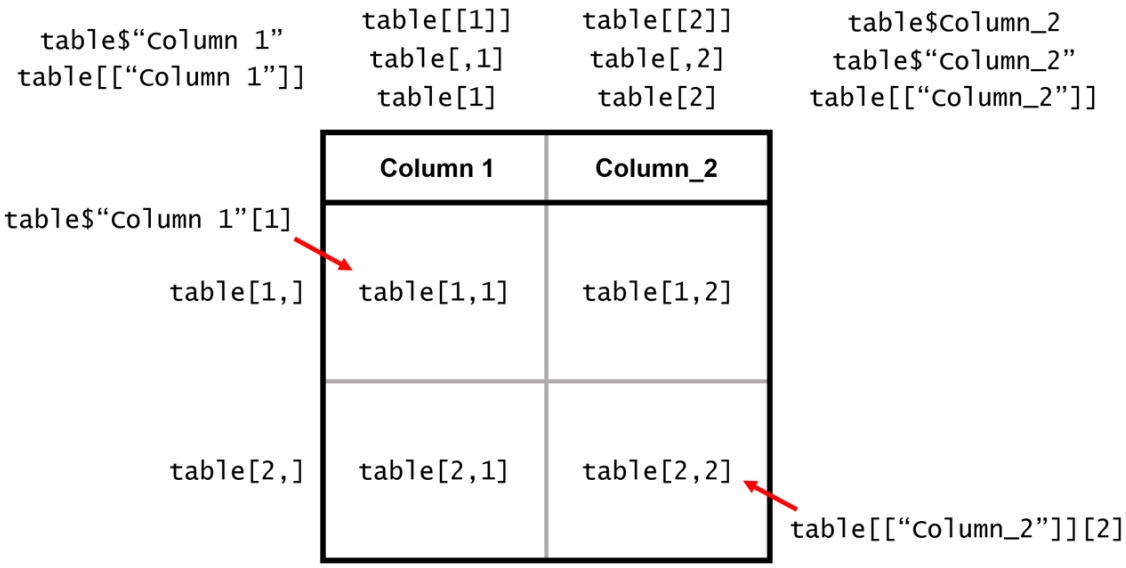

Figure 4. An example of how to index different things from a data frame called "table". Note that column 1 has a space in the name, as opposed to an underscore, which can present it's own difficulties and should be avoided. I have included it here however to show how versatile R can be and how we can overcome these potential issues

Conditional Statements

Now that we have seen all the basic data structures and operators of R, we move on to a fundamental series of components of almost all programming languages.

Conditional statements are a fundamental part of programming. They allow us to control the flow of a program by executing certain pieces of code only when specific conditions are met. In R, this means we can make our code respond differently depending on the values of variables, the outcome of comparisons, or the results of logical operations.

In everyday life, we make decisions based on conditions all the time: *If it is raining, I will bring an umbrella. Otherwise, I won't.*. In archaeology we might be more inclined to use conditional statements to ask if a certain criteria is met for determining

a species or typology, or if a value lies above a given threshold. The basis of conditional statements are the logical operators that we saw before, however here they are put into practice to then condition whether parts of our code are run or not.

In R, we express these kinds of decisions using `if`, `else if`, and `else`. These allow us to write code that says, "Do this if something is true, otherwise do something else."

Beginning with `if`, the basic R syntax consists of typing `if`, followed by a condition enclosed in parentheses `( )`, and a block of code within curly braces `{ }` that will be executed only if the condition evaluates to `TRUE`. The condition must be a logical expression that returns a single `TRUE` or `FALSE` value.

Most often, this logical expression takes the form of a comparison (such as `x > 0`), but R will accept any expression that evaluates a single logical value. For example, one could write `if (TRUE)` or use a logical variable as the condition.

In R, it's important to use curly brackets `{ }` to indicate the block of code to run when the condition is met. Unlike some other programming languages where indentation alone can indicate code blocks, R relies on these braces to group statements together.

Information

Remember that indentations are the space that exists between the left margin of the code, and the beginning of where our line of code begins. Indentations are a fundamental component of clean code, so please use them, regardless of whether R requires them or not! - it's also good practice to get used to using them for when you might start exploring other programming languages that are unable to function without them.

In R indentations are not a requirement, but they certainly produce cleaner code when used. In R, the `{ }` are used to indicate what code belongs to which *scope* (block of code).

Bellow is an example of a simple use of an `if` statement;

```
x <- 5

if (x > 0) {

  print("x is positive")

}
```

First we define `x` to contain the number 5, and then check if it is higher than 0. Given that the statement we are checking for within the `if` statement returns `TRUE`, then the code will run. If not, then the code will not run. This is the simplest form of an `if` statement possible.

In some instances, we may wish for something to happen if the criteria is *not* met. This is called an `else` statement. `else` statements are incredibly simple, all we have to do is provide the term `else` after the closing of the `if` statement's scope, followed by another block of code defining what will run if the statement is not met.

```
x <- 5

if (x > 0) {

  print("x is positive")

} else {

  print("x is negative")

}
```

Finally, there may be instances where we wish to compare multiple conditions. For example, if the first criteria we check for returns `FALSE`, check one other condition before reaching the `else` part of the code. This is known as an `else if` statement. An `else if` is literally as though we were joining and `else` statement, and an `if` statement together, because the syntax is the same;

```
if (x > 0) {

  print("x is positive")

} else if (x == 0) {

  print("x is 0")

```

```
} else {

  print("x is negative")

}
```

In this example, R will evaluate each condition in turn until it finds one that is `TRUE`. Once it does, it executes the corresponding code and skips the rest. If it doesn't find a condition that is `TRUE`, then it will run the code within the `else` block of code.

Information

Conditional statements form the building blocks for decision-making in your R scripts. As we move forward, you'll see how they are often used inside loops and functions to create flexible and powerful programs. You may feel that this is a bit abstract at the moment, but we will end up using these types of statements quite a lot as we continue forward. At the moment, all you need to understand is the syntax, and what a conditional statement is, and you will quickly pick up on where to use them and where not.

Loops

Loops are quite literally a means in which we loop over a series of operations. These are extremeley useful and are used in almost everything we do.

In programming, loops are control structures that allow us to repeat a set of instructions multiple times. This is especially useful when we need to perform repetitive tasks, such as applying the same calculation to each element of a vector or matrix, or running simulations multiple times. The value of loops, beyond their evident functional value, is how they also relate to clean code in that they allow us to take something repetitive, which would result in hundreds of lines of code, and reduce them to just a few lines. We will see this a bit more later.

In R, we have two primary types of loop, the `for` loop, and the `while` loop. While other programming languages have other additional types of loop, these aren't included in R. The only extra type of loop in R is the `repeat` loop, however this is certainly not used as much as the other two.

Each loop type is used in slightly different contexts, but they all share the same underlying purpose; repeatedly execute code until a condition is no longer met, or a set of elements is fully processed.

`for` Loops

`for` loops are the most common types of loops all programmers will use. The basic syntax of a `for` loop begins with the `for` keyword, followed by a rule inside parentheses `( )` that tells R how many times to run the loop, or what sequence to iterate over. This is followed by a block of code enclosed in curly brackets , which defines the instructions to execute on each iteration.

Consider a trivial example where we want to print the numbers from 1 to 5 on the screen. We can do this using;

```
x <- 1
print(x)
x <- x + 1
print(x)
x <- x + 1
print(x)
x <- x + 1
print(x)
x <- x + 1
print(x)
```

The way we could do this more efficiently and programmatically using a `for` loop is as follows;

```
for (i in 1:5) {
    print(i)
}
```

which would result similarly in;

```
>> [1] 1
>> [1] 2
>> [1] 3
>> [1] 4
>> [1] 5
```

There are a lot of things to unpack with this example. Hence why we start with something very trivial, before we move onto more complex (and of course useful) uses of loops. First of all let's look at what the code is actually saying. If we were to translate this into human language, it would go something like;

For every iteration in 1 to 5, print the value of the iteration on the screen

I have placed in blue the literal components of this phrase that is being done by R given the code we provided. The `for` loop knows it has to run the loop for iterations from 1 to 5 through the `1:5` component of our statement. The part of the code that states `i in 1:5` means that in each iteration of the loop, the variable `i` takes on the next value in the sequence from 1 to 5. This is why we can use `print(i)` — `i` is defined within the loop's scope and updated automatically each time.

Handy Tip

Please note that R, unlike some other languages, will not remove `i` from memory once the loop has finished running. While this might not necessarily be an issue, it can get pretty messy to handle so many things if the environment is filled with variables that are only used during a loop. One way to overcome this is to remove `i` from memory the moment the loop is over. We can do this by using the remove function; `rm()`. If we type `rm(i)` after the loop is over, `i` will no longer be in the memory of the R session. To not make a mistake, consider simply the following; `for (i in 1:5) print(i); rm(i)`, i.e., make sure that `rm(i)` is run AFTER the loop has run. One exception to this is in a function, but we will get to that later.

Handy Tip

In most examples you will find `for` loops using `i` as the main variable that is updated with every step of the loop. This however goes against what we have said on countless occasions about clean code. `i` is not at all informative about what we are doing, in many cases at least. `i` is often used because in mathematics it is very common notation to write something like x_i, however `i` is often overused for every loop regardless of whether we are doing mathematics or not. `i` is also kind of like the word "iteration", but I prefer to actually type out the entire word `iteration` to avoid ambiguity. It is good practice, therefore, to give a meaningful name to the variable we are defining within the scope of our loop.

Just to ensure that this is as clear as possible, the `for` loop syntax requires that we define within the parentheses immediately after the `for` keyword, a condition that will update itself with every iteration of the code contained within the block of code. The word `in` is a really important part of this as well. The `in` keyword is what tells R to assign each value from the sequence on the right-hand side to the variable on the left-hand side. Without it, the loop wouldn't know how to progress through the values.

It is important to note that a loop does not always have to go through sequences of integers, as we have seen so far with the example of `1:5`. We can use a for loop to iterate across any vector, with any type of variable, including characters, particular sequences, or even more complex structures like lists.

Suppose we have a vector of species names, we can fun a loop over each of these like follows;

```
species <- c("H_habilis", "H_ergaster", "H_erectus", "H_neanderthalensis", "H_sapiens")

for (individual_species in species) {

    print(individual_species)

}
```

In this case we have a vector of characters called `species`, we loop over the vector, defining the variable `individual_species`, and with each iteration of the loop, `individual_species` is updated to contain the contents of the next individual in the vector `species`

The final example I'll provide shows a bit more of a complex mathematical notion, just to show an example where we are not always simply printing onto the screen a bunch of arbitrary numbers. Imagine we have a mathematical formula, that is simply $y = 5x + 3$, i.e. a simple line, and we want to know all the values of y from a sequence of numbers between 0 and 1 at intervals of 0.25. We can do this by;

```
for (x in seq(0, 1, by = 0.25)) {

    y <- (x * 5) + 3
    print(y)

}
```

```
>> [1] 3
>> [1] 4.25
>> [1] 5.5
>> [1] 6.75
>> [1] 8
```

`while` Loops

So far we have seen that `for` loops are incredibly useful when we have a fixed set of values that we want to iterate through. There are cases, however, when we may not know how many times we need a loop to run. One way of doing this is to run a loop progressively while considering whether a condition is true or not. This is where `while` loops come in.

A `while` loop is kind of like a combination of a `for` and an `if` statement, to the point where we can literally program one by hand in this way, however this would be relatively counter productive. `while` statements follow the same syntax as `for` in as much that you first type the `while` keyword, followed by a rule inside parentheses that tells R how long to run the loop for, and finally a block of code informing R what we want to run. In this case, however, `while` accepts a boolean, as opposed to a sequence of values, and while the boolean is `TRUE` it will continue to run, however the moment the boolean becomes `FALSE`, the loop will end.

Let's provide another trivial example where we want to keep doubling a number until the value is larger than 100. We can do this using;

```
while (x < 100) {

    print(x)
    x <- x * 2

}
```

```
>> [1] 1
>> [1] 2
>> [1] 4
>> [1] 8
>> [1] 16
>> [1] 32
>> [1] 64
```

`while` loops are very useful, however are not as common. You also have to be careful that `while` loops know when to end, because otherwise they can run forever, an infinite loop, or get the computer to crash.

If you ever need to stop a loop manually inside of the code, you can use a `break` statement, however we will introduce what `break` is in the next type of loop.

`repeat` Loops and the `break` function

While `for` and `while` loops give us control based on a fixed sequence or a condition, there may be cases where neither structure feels quite right — particularly when we want a loop that runs indefinitely until we decide it's time to stop it. This is exactly the role of a `repeat` loop.

A `repeat` loop is the most barebones looping structure in R. It does not require a condition to begin with. Instead, it simply keeps running the code inside it forever, unless we explicitly tell it to stop using the `break` command.

The basic syntax looks like this;

```
repeat {

    # some code

    # evaluate a condition to determine when to stop

    if (some_condition) {

        break

    }

}
```

I am yet to find a use for this type of loop in archaeology, however it is useful to know of its existence, and more importantly, the `break` command that can be used to stop any loop prematurely if necessary.

Functions and Object Oriented Programming

Object-oriented programming (OOP) and function-oriented programming (FOP) are fundamental components of computational theory. For this reason, I've set aside a separate section of the guide to explore these two important paradigms more closely.

Until now, we've been writing code using a step-by-step, linear approach. While this works well for small tasks, as projects become more complex—and especially more *repetitive*—we need new tools to keep our code clean, readable, and efficient.

These concepts can be difficult at first, especially if you're not from a programming background. But I think it's useful to at least be aware of what they are. If you're here primarily to use R for your own studies and not to become a full-time programmer, feel free to skip straight to the "Functions" section of this document. The key takeaway is this: if you find yourself writing the same code multiple times, that's a good sign that it's time to use a function.

Handy Tip

Remember one of the first points we came across with regards to clean code; *Functions should be small, and do only one thing*, which i later refined to **Functions should be small, and do a limited handful of useful things**. This is something we need to keep in mind here. People can often overuse functions when it makes little or no sense to do so. Likewise, if you find yourself repeating the same code over and over again, this is a sign that you need to consider a different approach. Efficient programming consists in using a bunch of functions, that can then be called upon, potentially inside another function itself, to clean up code where repetition is present. Classes, on the other hand, provide a different means of handling our code, that we will see later on.

Before we dive in, let's take a quick look at two major programming paradigms: Function-Oriented Programming (FOP) and Object-Oriented Programming (OOP). These represent two different ways of thinking about how we write code, and many programming languages—including R—support both to varying degrees.

The Function-Oriented Programming Paradigm

As the name suggests, FOP focuses on writing and using *functions* to handle data and perform tasks. Functions are conceptually simple: given the same input, they always return the same output, and they don't affect anything outside their scope (this is called having "no side effects").

FOP emphasizes *immutability*: the data passed into a function is not changed, and the function itself doesn't modify the outside world. This might seem abstract at first, but it becomes clearer with experience.

Languages that favor FOP include Haskell (a pure functional language), as well as multi-paradigm languages like Python, JavaScript, and R. However, R is somewhat unique: it appears functional on the surface, but many of its deeper mechanisms and packages are built around OOP principles.

The Object-Oriented Programming Paradigm

OOP organizes code around *objects*, which are instances of *classes*. A class is like a blueprint, and an object contains both *attributes* (data) and *methods* (functions that operate on the data). We'll explore this further when we get to S3 and S4 classes in R.

OOP emphasizes concepts like *polymorphism* (different types responding to the same method name in different ways) and *inheritance* (classes building on other classes). Objects can maintain a state over time, and methods can change an object's internal data.

Many lower-level programming languages are built primarily around OOP. Java and C# are almost exclusively object-oriented, while C++ supports both paradigms but is often used with classes. Python, like R, supports both approaches and is especially beginner-friendly from an OOP perspective.

As for R—the focus of this guide—things are a bit more nuanced. While R code often looks functional on the surface, Base R is built to support OOP. Many functions you use daily (like `print()` or `summary()`) are actually examples of OOP concepts, using what are known as generic functions and method dispatch.

Functions

The basic syntax or skeleton of a function is as follows;

```
name_of_function <- function(input1, input2 = NULL, ...) {

    # some code

    return(output)

}
```

In R, we create a function and assign it to a name using the `<-` operator. This allows us to call the function later using that name. We also declare that it is a function using the keyword `function`. This is then followed by parentheses, where we define (if any) the input parameters the function requires to work, and then `{ }` where, just like before, we write the code we wish to run when the function is called. A very important extra component here is the keyword `return` that appears at the end. `return` is a very important component of a function, as it defines what the output will be, but is also something that we can use to break out of the function. Using a `return` however is not obligatory, many functions might be used to not produce an output, so we are not forced to have the `return` in all of our functions.

⚠ Warning

`return` can be both useful and not, if we use it incorrectly. The moment the function reaches `return`, the function will end. If this is the desired goal, and the `return` function is nested inside a conditional statement, for example, then great. However, you have to be careful with where you put a `return` statement, because it could have adversarial effects if you use it incorrectly.

While the above example is the general skeleton, functions are actually incredibly versatile, and can be as simple or as complicated as we would like. Note in the previous example of syntax that I put two inputs, followed by an ellipses, "...", and that `input2` is equal to `NULL`. I wrote the skeleton in this way on purpose to remind the reader that (A) we can define as many inputs as we want, and (B) we can also predefine what some of them are, which is a very valuable attribute of functions.

Let's go through this bit by bit with some examples. Let's take an example of a function that is used to take two numbers, and add them together.

```

addition <- function(number1, number2) {

    number1 + number2

}

addition(number1 = 1, number2 = 2)
```

```
>> [1] 3
```

Note that I didn't include `return`, however the function still gave an output. I did this on purpose just to provide an example of how versatile R is, but not always for the better. This is not a very good example of clean code because it is not immediately available to the user what the output is. In this case, the output is simply the last line. However, if we had a huge function, this might be a bit harder to see. It is good practice to just get used to always using the `return` function to ensure that the reader of your code clearly knows what the output of the function will be.

A much cleaner version of the same function would be;

```

addition <- function(number1, number2) {

    answer = number1 + number2

    return(answer)

}

addition(number1 = 1, number2 = 2)

addition(3, 10)
```

```
>> [1] 3
>> [1] 13
```

In this example I introduced another version of calling the function, which is what I'd like us to focus on now. In the first instance of using the function, inside the function call, we specified that `number1 = 1`, and that `number2 = 2`. This is one way of passing the two inputs into the function by explicitly specifying what each input is. In the second example, we simply introduced `3` and `10` without specifying whether `3` was `number1` or `number2`, and the same for the number `10`. There is no right or wrong here, R will assume that the first value you pass will correspond to the first input, and so on, however, it can often be very useful to explicitly define what parameters you are using. This becomes relevant in a number of examples, where predefined values are introduced.

Information

Simply note in the above example how we used the `<-` operator to declare the name of the function, however inside the function, `answer` is stored using `=`. We could very easily use both `=` and `<-` interchangeably, however, the general recommendations of R are to use `=` inside of a function, and `<-` outside. While I personally do not always follow this convention, it *is* recommended, so I have tried to do so here to promote adopting good habits early on.

Imagine we want to create a function that calculates an incredibly simple polynomial, where in the majority of cases we want to simply calculate x^p, where p is usually 2, but in some instances we might wish to change this. This would be a good example of introducing a default parameter.

```

simple_polynomial <- function(x, p = 2) {

    answer = x^p

    return(answer)

}

simple_polynomial(x = 2)
simple_polynomial(3)
simple_polynomial(4)
simple_polynomial(5, p = 3)
simple_polynomial(5, 3)
```

```
>> [1] 4
>> [1] 9
>> [1] 16
>> [1] 125
>> [1] 125
```

In this case, when we pass into the `function` a parameter that is already defined as something else, in this case `p = 2` the moment the function is declared, that means that the function will automatically assume that `p = 2`, unless stated otherwise. In R, if you explicitly name your parameters in the function call (e.g., x = 3), the order doesn't matter. But if you rely on positional arguments (like just writing 3, 5), the order becomes essential. We could type `simple_polynomial(p = 3, x = 2)` and it would still work, but this is not really recommended, and in other programming languages would produce an error. It is also true that in R we can also provide inputs in the wrong order. For example;

```

addition <- function(number1, number2, number3) {

    answer = number1 + number2 + number3

    return(answer)

}

addition(3, c = 2, b = 1)

addition(b = 3, a = 2, 1)
```

would all work, however, this is incredibly messy and very problematic if you then move to other programming languages like Python that do not allow this to happen. From this perspective, it is always recommended that you write code as clearly as possible, and try and follow the code in a relatively linear way to avoid future issues. While R may be very flexible, this is a blessing and a curse, and it is best to get rid of bad habits like that now, before you move onto another language where these types of function calls would produce errors and can be very frustrating.

Information

R allows arguments to be supplied in any order as long as they are named explicitly.

While this is the general structure of a function, there are a bunch of other things we could talk about, some of which I think it is worth mentioning here.

Functions can return multiple objects by using lists

As stated earlier in this document, lists are a really common component of R, yet might be confusing at first. Nevertheless, the vast majority of R functions return lists, as opposed to single values, therefore it is worth mentioning this here. Consider the definition of a function that will simply return some summary statistics for us;

```
basic_stats <- function(x) {

    mean_value = mean(x)
    standard_deviation_value = sd(x)

    return(
        list(
            mean = mean_value,
            standard_deviation = standard_deviation_value
        )
    )

}

basic_stats(c(1,2,3,4,5))
```

```
>>
   $mean
   [1] 3

   $standard_deviation
   [1] 1.581139
```

Be careful of a function's scope!

Scope is not that important in R for some things, but it is bad practice to abuse this behaviour. To remind the reader, the scope of a function is everything that exists between the `{ }` brackets. In languages such as JavaScript, the scope is highly important and you cannot access variables across all scopes. In R this is true to some extent, but not to all. Consider the following example;

```
x <- 2

simple_polynomial <- function(p = 2) {

    answer = x^p

    return(answer)

}
```

This would work, because a function has access to the global environment, therefore even though `x` is not passed into the function, it would still know what `x` is. From another perspective, and tying into one of the things we saw before with loops;

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

sum_function <- function(x) {

    result <- 0
    for (value in 1:length(x)) {
        result <- result + x[value]
    }

    return(result)

}
```

The loop declares a variable called `value`, however this variable is only available in the scope of the function, and would not be available to us once the function has finished running. Earlier we stated that you have to use `rm()` to remove `i` from

memory after a loop has finished running, although here this would not be necessary. The moment `sum_function` has finished, the variable `value` is automatically removed.

Don't trust people!

An odd one, but incredibly relevant. When defining parameters of a function, or any software for that matter, one thing we have to be incredibly careful with is the idea that no matter how well we document a function, it is likely that the end user will not even read the documentation. From this perspective, it is very common for people to misuse functions, which can require some effort to avoid bigger problems. For example, if a function requires a number to be passed in as one of the inputs, but a string is passed through instead, this will produce an error. Likewise, if the format of something is not the right type, this will also produce an error. To avoid headaches of either (A) being contacted saying that something you wrote doesn't work but just because it wasn't being used right, or (B) being able to not use a function yourself and not being able to remember why, we can add a series of conditional statements, warnings and our own personalized errors to inform the user of the mistake they might be trying to make.

If we go back to the example of a function that calculates a simple polynomial from before;

```

simple_polynomial <- function(x, p = 2) {

    answer = x^p

    return(answer)

}
```

and we try to pass in as x "this is not a number", we will get an error;

```
>> Error in x^p : non-numeric argument to binary operator
```

First of all, if we know well enough the general syntax of R, we should be able to deduce ourselves through the term "non-numeric argument" that we have done something wrong by passing in a string. Nevertheless, some of these errors can be quite cryptic, even to the well-trained coder. One thing we can do is create a conditional statement that checks if x is actually a string, and prevent the user from making this mistake, while providing a much more personalised, easy to read message.

```

simple_polynomial <- function(x, p = 2) {

    if (!is.numeric(x)) {
        stop("x must be a number!)
    }

    answer = x^p

    return(answer)

}

simple_polynomial("this is not a number!)
```

```
>> Error in polynomial("this is not a number") : x must be a number!
```

Handy Tip

Note that we have used in this one example, all of the theory and building blocks we have seen up until this point. Using `is.numeric()` we checked if `x` is a number or not. Using `!` we see if the input is *not* a number, and using that as the input to an `if` statement, we then produce an error using the `stop()` function, that allows us to stop the function from going any further (like if we were to have used `return`), and we have provided a meaningful message to inform the user what they have done wrong.

Finally, imagine that an input is allowed, but likely to cause an error, or make some mistake, but is not something we necessarily want to stop all together. In the previous example, we checked whether x is numeric, but say we want to ensure that p is at least higher than the number 1. We can do this using;

```r

simple_polynomial <- function(x, p = 2) {

    if (!is.numeric(x)) {
        stop("x must be a number!")
    }

    if (p <= 1) {
        warning("it doesn't make much sense to use a p value of 1!")
    }

    answer = x^p

    return(answer)

}
```

By doing this we produce a warning message, but the code will still run even if p is less than or equal to 1.

Object Oriented Programming

Coming soon...

S3 Classes

Coming soon...

S4 Classes

Coming soon...

Approaching Mathematics Programmatically

Throughout this document, we've explored all the building blocks necessary to become efficient R programmers and to develop a solid foundation for understanding how programming works more broadly. At least, I hope we have.

This next part of the document is intended to demonstrate how we can leverage these tools for mathematical computation — something that archaeologists often shy away from, but which doesn't have to be *that* intimidating.

The computer is one of the most powerful calculators available to us today. It can perform thousands — even tens of thousands — of calculations in the blink of an eye. What's more, it is far less prone to error than we are; and when a mistake does occur, it is almost always the programmer's fault, not the machine's. Another essential benefit is automation: computers can carry out complex or repetitive tasks that would otherwise be impractical or prohibitively time-consuming by hand.

In today's academic landscape, it is common for mathematicians and computer scientists to share courses and collaborate closely. This reflects a deeper reality: modern mathematics is rarely performed entirely by hand. While university curricula often require students to carry out lengthy calculations manually, this is primarily to ensure a solid grasp of the underlying concepts and theory.

In actual practice, mathematicians, scientists, and engineers almost always use computers to perform the bulk of their computations. The manual exercises serve as a training ground — a way to internalize principles and develop intuition — but when it comes to real-world problems, computational tools are indispensable. This shift not only saves time but also opens the door to exploring much more complex and large-scale problems that would be impossible to tackle otherwise.

At its core, mathematics is about recognizing patterns, relationships, and logical rules — ideas that map naturally into programming concepts. Numbers become variables, equations become functions, and datasets become vectors or matrices. This correspondence means that any mathematical idea you can write on paper, you can express in code.

By programming these concepts, we unlock powerful advantages. Complex calculations that would take hours or days by hand can be done instantly. We can process large datasets effortlessly, something increasingly important in fields like archaeology where data is abundant but challenging to analyze. Furthermore, programming allows us to test mathematical ideas and visualize results dynamically, helping us gain deeper insights.

Don't be intimidated if it feels overwhelming at first. Like any new skill, programming mathematical computations takes practice and patience. Mistakes are inevitable, but each one is an opportunity to learn. The key is to start simple, experiment boldly, and build your confidence step by step.

What I would like to introduce here is the first steps towards trying to approach mathematics using programming. If we understand what the building block of some of the scariest functions are, and we already know the building blocks of the R syntax, maybe we can combine both to overcome some of these fears, and begin to work on these questions ourselves.

A quick first example

Let's begin with a quick example, and then dive into some more specific case studies.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

Have you seen this before? Even if it doesn't look familiar at first glance, it might surprise you to know that you probably use it all the time. This is the formula for the arithmetic mean — or, more simply, the average. While it uses some general rules of mathematical and statistical notation, once you understand those, it's surprisingly straightforward to translate into code.

In R, you could calculate the arithmetic mean simply using the built-in `mean()` function. But rather than relying on shortcuts immediately, it's worth taking the time to understand what the computer is doing behind the scenes. Rebuilding basic formulas like this from scratch is a great way to deepen both your programming and mathematical intuition.

First of all, μ is the greek letter *mu*, which is commonly used to represent the mean in statistics. The next potentially intimidating symbol is the large captial Σ, or *Sigma*. This is the summation symbol - it tells us to add up all the values of whatever comes next, which in this case is x. Below *Sigma*, we see $i = 1$, and above it, n. n is commonly used to express the total length or number of observations. i is usually used to depict the index, a way of saying "go through each element one by one". What this is telling us, is that we have to calculate the sum of all values of x at position i (which is literally what x_i means), from 1 ($i = 1$), to n. After summing all of x, we multiply the result by $\frac{1}{n}$ - or in plain language divide the total by the number of items.This is exactly the same average you probably learned to calculate back in primary school.

Hopefully at this point you may have already begun seeing what different programming skills can be applied to performing parts of this equation. If you haven't, don't worry. We'll walk through this first step together now.

First of all, x is a vector, therefore we can define x by using the vector data structure format, or simply concatenate a bunch of numbers together using `c()`. Following this, the idea of adding each of the values together requires that we index each value of x at position i, using our knowledge on indexing. To go through each value, we can use a `for` loop. Finally, knowing the mathematical operators, the rest should be relatively straight forward. Because this is a formula we are likely to use a lot, it would be wise to store it in a `function`, as opposed to writing everything out for each time we need it. Let's go through this together;

```

my_first_formula <- function(x) {

    n <- length(x) # first we calculate n, by calculating the length of x
    Sigma <- 0 # we start with the sum of all values at 0

    for (i in 1:n) {

        Sigma <- Sigma + x[i] # for each value of x at index i, we add it to the sum

    }

    answer = Sigma / n # or Sigma * 1/n

    return(answer)

}
```

A good way to check that this is working as it should would be to actually compare it with a known function that works - a form of ground truth (this is what I do every time I work with these types of things!).

```

my_result <- my_first_formula(c(1,2,3,4,5))
r_result <- mean(c(1,2,3,4,5))

print(my_result)
print(r_result)

if (my_result == r_result) {print("You did it!")}
```

```
>> [1] 3
>> [1] 3
>> [1] "You did it!"
```

Handy Tip

`x` is used as input to the function, and has to be a vector, however for simplicity i have not included a conditional statement here to ensure that `x` is a numeric vector. This is definitely something that I would normally do, but to begin with, we can start like this. Just remember to try and make sure people can't use the function wrong. Another useful thing to do here could be to ensure that `NA`s are not present in the sample, because that would be another cause for error. Either way, this version of the function is relatively functional already.

Another quick example - the Euclidean distance

Let's look at another small example, just to continue to get familiar with programming and doing mathematics in this way. The Euclidean distance is a formula that I use every single day, without a doubt. This formula is at the heart of many real-world applications — from measuring distances between landmarks in geometric morphometrics, or the height of something off the ground, or the distance between two caves. For this reason, it's a pretty useful example to work with.

The distance between two points, with x and y coordinates, using the Euclidean formula is very simply;

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Remember, the little number at the bottom of x in x_1 literally means the x coordinate of the first point, while x_2 is the x coordinate of the second point. This is already incredibly easy to calculate. In R the square root can be calculated using the `sqrt()` function, the square of a value can be calculated using $\hat{2}$, so we can very easily calculate this using;

```
1
2 point1 <- c(2, 3)
3 point2 <- c(5, 6)
4 sqrt((point1[1] - point2[1])^2 + (point1[2] - point2[2])^2)
```

```
>> [1] 4.242641
```

If we extend this to three dimensions, all we have to do is update the formula to include the z dimension. However, it is more common in mathematics (and in programming) to make a more generalisable approach. For this, let's take the *real* Euclidean distance formula, and unpack it like before;

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Thankfully we have already seen the summation operator before, so we know how to do that. The idea of p and q might seem a little bit more abstract, but its literally just taking the coordinate from the first point, p, and a coordinate from the second point, q, we can calculate the distance between the two using this formula. Here n is the technically the number of dimensions. If we had a two dimensional point, then this would be literally what we had before;

$$\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

And if we were in three dimensions;

$$\sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + (q_3 - p_3)^2}$$

Programming this should be relatively easy, given how efficient we were with the last example!

```r

my_euclidean_distance <- function(q, p) {

    n1 <- length(q) # get number of dimensions in q
    n2 <- length(p) # get number of dimensions in p

    # it's worth checking that q and p have the same number of dimensions:

    if (n1 != n2) {
        stop("The two points don't have the same number of dimensions!")
    } else  {
        n <- n1
    }

    Sigma <- 0 # we start with the sum of all values at 0

    for (i in 1:n) {

        coordinate1 <- q[i]
        coordinate2 <- p[i]
        squared_difference <- (coordinate1 - coordinate2)^2

        Sigma <- Sigma + squared_difference

    }

    answer = sqrt(Sigma)

    return(answer)

}
```

Here we first check how many dimensions each point has by using `length()`. Then we make sure they match — we can't calculate a distance between a 3D point and a 2D point! We then move on to calculating once again the sum of the squared differences in coordinate values. Finally we calculate the square root of that, and return the answer. For clarity, i also prefer to have a single version of `n` at the end as opposed to an `n1` and a separate `n2` at the end that might be mathematically confusing.

A dictionary of common mathematical symbols, and how to program them

Vector notation

A vector in mathematics is simply denoted by a symbol, say x, with a tiny arrow above it, for example $\vec{x}$. In notation they are typically written as follows;

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

However, they are more often written horizontally;

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

To program this into R, we simply use `c(x1, x2, x3)`.

Matrix notation

A matrix is simply a 2-dimensional arrangement of numbers, often used to represent data, systems of equations, or transformations. Like vectors, matrices are written with brackets, and the elements are typically denoted using two subscripts:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

This example shows a 3×3 matrix, meaning it has 3 rows and 3 columns. In general, the element a_{ij} is the value found in the ith row and jth column.

To program this into R, we use the `matrix()` function. For example:

```r

matrix(c(a11, a21, a31, a12, a22, a32, a13, a23, a33), nrow = 3, ncol = 3)
```

Note that R fills matrices *column-wise* by default, so we enter the elements in that order.

Matrix Operations

Some common operations on matrices include:

- **Transpose** (A^T): Flips the matrix over its diagonal, swapping rows and columns. In R, you can use `t(A)`.
- **Matrix multiplication**: For matrices A and B, the product AB is defined when the number of columns in A equals the number of rows in B. In R, use `A %*% B`.
- **Element-wise multiplication**: Multiply matrices element by element (only when dimensions match). In R, use `A * B`.
- **Normalization**: Normalizing a matrix can mean different things depending on context; a common approach is to scale each element by dividing by the matrix norm or the max/min element. For example¡;

```r
norm_A <- sqrt(sum(A^2))
A_normalized <- A / norm_A

```

f(x) - the mathematical function

One of the simplest things we will see in many mathematical formulas is $f(x)$. It feels nice here to just say, all $f(x)$ is, is the mathematical way of writing `function`. x is the input to the function, and whatever happens after that will be found in the rest of the formula. A common example is;

$$f(x) = \frac{1}{1+e^{-x}}$$

Which is the sigmoid function, and the basis of the logistic regression formula. Sigmoid returns a value between 0 and 1. e, as explained earlier, is euler's number which is a constant. To program the above function, we simply need;

```r

sigmoid <- function(x) {

    f_x <- 1 / (1 + exp(-x))
    return(f_x)

}
```

Summation

This is the first element we have seen, and one of the most common symbols we will come across.

$$\sum \quad \sum^{n} \quad \sum_{i=1}^{n} \quad \Sigma_{i=1}^{n}$$

This symbol simply means that we have to sum up everything that goes after it. We can see this symbol written in a number of different ways, but the most common is the third example above, where you have an indication of what number to start from, $i = 1$, and then to what number you go to, n. For example $\sum_{i=1}^{n} x_i$ would mean to sum all values of x from 1 to n.

In R we can easily program this using the `sum()` function, however if you need to control the calculation a bit more, a `for` loop is also perfect.

Product

This is very similar to the Σ symbol, however uses multiplication instead of summation.

$$\prod \quad \prod^{n} \quad \prod_{i=1}^{n} \quad \Pi_{i=1}^{n}$$

This symbol simply means that we have to multiply everything that goes after it. We can see this symbol written in a number of different ways, but the most common is the third example above, where you have an indication of what number to start from, $i = 1$, and then to what number you go to, n. For example $\prod_{i=1}^{n} x_i$ would mean to multiply all values of x from 1 to n.

In R we can easily program this using the `prod()` function, however if you need to control the calculation a bit more, a `for` loop is also perfect.

Derivatives

Derivatives and integrals are a fundamental component of calculus, and are often quite scary to confront and look at. This small document is not meant to be a solution to all of your issues in mathematics, so I cannot go into too much detail about what these things are, but I can certainly tell you that very few people today actually calculate these things by hand, and R is a powerful tool to help us with that.

A derivative tells us how fast something is changing. In mathematics, the notation might look like:

$$\frac{d}{dx}f(x)$$

although we can also see $f'(x)$, Δ, or ∂. Their use and function depends quite a lot on what they are being used for. As we have already seen, $f(x)$ is the same as a function. So any declaration of $f(x)$ can be solved by simply programming a function that computes the formula in question.

We hardly ever calculate derivatives by hand anymore, but what we do instead is try and approximate them using numerical methods. In R there is a built-in function, `D()`, that accepts an expression as input, along with the variable we wish to differentiate. Here's an example using the function $f(x) = x^2$:

```
1
2 expression <- expression(x^2)
3 D(expression, "x")
```

This would return $2x$, which is the derivative of x^2. In this snippet of code, you would have noticed the `expression` function being used as opposed to actually defining a `function`. This is because the function is being declared symbolically.

If you instead wanted to approximate the derivative numerically (as is often done when you have values, but not a symbolic expression), you could write a small function like this:

```
1 f <- function(x) x^2
2 df <- function(x, h = 1e-5) (f(x + h) - f(x)) / h
3 df(2)
```

This tells us the rate of change of x^2 at the point $x = 2$, which would be close to 4.

As archaeologists, we will likely not need to compute many of these ourselves — but it is very helpful to know what the symbols mean and how we could approach them with code when needed.

If you truly wish to dive into this, however, I would recommend learning the basics of calculus, because this is where it definitely begins to get tricky.

Integrals

Integrals, much like derivatives, are not usually calculated by hand, however they are incredibly useful and are certainly something I have seen quite a lot in archaeological literature when using mathematics.

Integrals are written using the integral symbol;

$$\int \quad \int_a^b \quad \int_0^\infty \quad \int_{-\infty}^\infty$$

The integral is normally used to compute the area under a curve of a function, or helps us approximate this area numerically. The key components of how integrals are written are similar in part to the summation and product symbols (Σ and Π), however it is more often that we also see this associated with the infinity symbol ∞. The lower and upper values indicate the bounds of the function, and literally tell us what range we have to integrate over. For example in $\int_0^1$ we would integrate the function from the values 0 to 1. The reason we see ∞ a lot in literature is that these are theoretical functions, and simply state that they could technically be integrated anywhere, although of course when we compute them we never actually do that.

Let's take a quick example;

$$\int_a^b f(x)dx$$

What this is telling us is that we have to integrate from a to b the given function. In this specific formula, $f(x)$ has not been defined, so we'd need to have some context as to what we are integrating, however let's assume that it $f(x) = x^2$. dx, on the other hand, lets us know that we are integrating with respect to the change in x; d indicates the rate of change, while x is the variable we are observing this rate to.

In R there is a function called `integrate()`, where we can pass a function in and define its upper and lower limits. Say we wanted to integrate the above formula between 0 and 1, we would do so through;

```r

f <- function(x) x^2
integrate(f, lower = 0, upper = 1)
```

```
>> [1] 0.333333 with absolute error < 3.7e-15
```

There are many ways to approximate integrals, and the method we use often depends on the number of dimensions involved. So far, we've looked at single-variable integrals, but integrals are not limited to just one axis — in many applications, especially those involving surfaces or volumes, we need to integrate over two or more variables.

Let's take a real-world example from surface texture analysis in archaeology. The following formula calculates a surface roughness measure known as *Sq* (Root Mean Square height):

$$Sq = \sqrt{\frac{1}{A}\int\int_A z(x,y)^2 dxdy}$$

At first glance, the double integral might seem intimidating. But don't worry — it just means we are integrating in two dimensions, across both the x and y directions of a surface.

Here's how to break it down;

1. $z(x,y)$ is a function that gives us the height of the surface at each point (x,y). This function is obtained after the surface has been leveled and centered.
2. $z(x,y)^2$ means we have to square this height - this is a common thing in mathematics to avoid negative values.
3. the double integral $\int\int_A$ means we are integrating the squared heights across the whole surface A. dx and dy simply mean across and with respect to the change across the x and y axis. In this case, integrating this specific surface simply requires that we sum the values!
4. Finally dividing by A gives us the mean, while the square root puts the value back into the original units of height (μm).

How do we know we have to use the sum? Why did the formula not just use Σ instead of $\int$? Well, this technically is because integration of a definitely defined function is similar to summing.

How do we work with this formula then and program it in R?

First we need to define dx and dy, which is the rate of change across the x and y axes. In surface analysis this is the distance between two points, and is provided directly by the microscope. If we do not have this, but we do have a vector of x and y coordinates. We can calculate dx as the difference, `diff()`, between two values. Or simply the difference between the first and the second x values. The same can be said of y. A is the area, which is simply all of the product of all the dimensions. The result is then the square root of the sum of these squared values, times by dx and dy, and finally divided by A.

```r

Sq <- function(x, y, z) {

    dx <- diff(x)[1]
    dy <- diff(y)[1]
    A <- dx * dy * length(z)
    result <- sqrt(sum(z^2) * dx * dy / A)
    return(result)

}
```

Piecewise Functions

Sometimes in mathematics, a function behaves differently depending on the value of a variable. These are called piecewise functions, and are often written using curly braces to define conditions;

$$f(x) = \begin{cases} x^2 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

This means, if x is greater or lower than 0, then $f(x)$ is x^2, however if x is lower than 0, then $f(x)$ is equal to -1. This evidently is the same as an `if` statement in R.

Trigonometry

Trigonometric functions are fundamental in mathematics and are widely used in various fields, including archaeology, physics, engineering, and data science. They relate the angles of a triangle to the lengths of its sides, and are essential for working with periodic phenomena such as waves or rotations.

The most common trigonometric functions are:

- Sine: $\sin(\theta)$
- Cosine: $\cos(\theta)$
- Tangent: $\tan(\theta)$
- Arcsine (inverse sine): $\arcsin(x)$ or $\sin^{-1}(x)$
- Arccosine (inverse cosine): $\arccos(x)$ or $\cos^{-1}(x)$
- Arctangent (inverse tangent): $\arctan(x)$ or $\tan^{-1}(x)$

These functions take an angle θ (usually in radians) and return ratios of side lengths in a right-angled triangle or, in the case of inverse functions, return an angle given the ratio.

For example, the sine function is defined as:

$$\sin(\theta) = \frac{\text{opposite side}}{\text{hypotenuse}}$$

In R, trigonometric functions are built-in and work with radians by default:

- `sin(x)` computes $\sin(x)$
- `cos(x)` computes $\cos(x)$
- `tan(x)` computes $\tan(x)$
- `asin(x)` computes $\arcsin(x)$ (returns radians)
- `acos(x)` computes $\arccos(x)$ (returns radians)
- `atan(x)` computes $\arctan(x)$ (returns radians)

If you need to convert between degrees and radians, you can use:

$$\text{radians} = \text{degrees} \times \frac{\pi}{180}$$

and

$$\text{degrees} = \text{radians} \times \frac{180}{\pi}$$

which in R can be done as:

```
radians <- degrees * pi / 180
degrees <- radians * 180 / pi
```

These functions are very useful for calculating angles, rotations, oscillations, and periodic patterns, which may appear in archaeological data such as orientation of artifacts or cyclic phenomena.

Beyond the functions mentioned above, another function exists that is slightly different, the `atan2` function.

Unlike the regular arctangent function $\arctan(x)$ that takes a single argument (the ratio of sides), `atan2` takes two arguments, usually representing the coordinates (y,x) of a point in the Cartesian plane.

It returns the angle θ between the positive x-axis and the point (x,y), measured in radians, and correctly handles the signs of both arguments to determine the correct quadrant of the angle.

Mathematically:

$$\theta = \text{atan2}(y,x)$$

In R, this is implemented as `atan2(y, x)`.

This is especially useful for converting Cartesian coordinates to polar coordinates or for determining the direction angle between two points.

Greek Letters

Greek letters are commonly used in mathematics, statistics, physics, and many scientific disciplines. Below is a table of the most commonly used Greek letters in both lowercase and uppercase forms:

Lowercase	Uppercase	Name
α	A	alpha
β	B	beta
γ	Γ	gamma
δ	Δ	delta
ε	E	epsilon
ζ	Z	zeta
η	H	eta
θ	Θ	theta
ι	I	iota
κ	K	kappa
λ	Λ	lambda
μ	M	mu
ν	N	nu
ξ	Ξ	xi
o	O	omicron
π	Π	pi
ρ	P	rho
σ	Σ	sigma
τ	T	tau
υ	Υ	upsilon
ϕ	Φ	phi
χ	X	chi
ψ	Ψ	psi
ω	Ω	omega

The relevance of these symbols in mathematics and statistics is often huge. While many of these symbols are simply used to denote variables, and might not have much of a deeper value. Some others do have a large mathematical use. For example, π is likely the most famous, being the constant of 3.14159, which is already a reserved word in R as `pi`. In alphabetical order following that, α and β are very often referred to in linear algebra as coefficients for a line, e.g. $f(x) = \alpha x + \beta$. Next δ can often be found to refer to things like the rate of change, especially in it's upper case format Δ, which can simply be calculated as the difference between two values `x[2] - x[1]`. ε is very frequently found in the context of noise, or some form of perturbation. Γ is very often used in engineering as the symbol for a rotation matrix. θ is very frequently used to symbolise angles. μ and σ are incredibly common in statistics as they represent the mean and standard deviation of a distribution. ρ, τ and κ are very frequently used as coefficients. Finally χ is a famous symbol that refers to a specific probability distribution.

Beyond these, another very common letter or symbol based coefficient is euler's number, e, which is stored in R using the function `exp()`.

Other symbols and operators

Bellow are a collection of a bunch of other very common symbols that appear in mathematics that may not be as familiar, but have R equivalents.

Table 2. Logical Operators, Set Notation, and Comparison Symbols with R Equivalents

Mathematical Symbol	R Equivalent	Description
$=$	`==`	Equality test
$\neq$	`!=`	Inequality (not equal)
$<$	`<`	Less than
$>$	`>`	Greater than
$\leq$	`<=`	Less than or equal to
$\geq$	`>=`	Greater than or equal to
$\wedge$	`&&`	Logical AND (single value)
$\vee$	`||`	Logical OR (single value)
&	`&`	Element-wise AND (vectorized)
$\mid$ or $\vee$	`|`	Element-wise OR (vectorized)
$\neg$	`!`	Logical NOT
$\in$	`%in%`	Set membership
$\notin$	`!(%in%)`	Not in set
$\subseteq$		Subset (no direct R operator)
$\cup$	`union()`	Set union
$\cap$	`intersect()`	Set intersection

Table 3. Probability and Statistical Operators with R Equivalents. Probability and expectation have not been included because this can highly depend on context

Mathematical Symbol	R Equivalent / Function	Description
$P(A)$		Probability of event A (estimated by mean of logical)
$\mathbb{E}[X]$		Expectation (mean) of random variable X
$\mathrm{Var}(X)$	`var(x)`	Variance of random variable X
$\mathrm{Cov}(X,Y)$	`cov(x, y)`	Covariance between X and Y
$\mathrm{Corr}(X,Y)$	`cor(x, y)`	Correlation between X and Y

References

1. Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship* (Prentice Hall, New Jersey, 2009).