iWOMEN in ARC

Introduction to programming in R

Day 1

Introduction to programming in R
March 2025

Digital Research Alliance of Canada | Alliance de recherche numérique du Canada

Today’s agenda:

Part 1	Part 2
What is programming?	Data structures and data types
What is RStudio and why use it?	Operators and functions
How to write/run codes in RStudio?	Loops and conditions
What are the building blocks of a code?	How to use pre-written functions (packages)?

What is programming?

- Programming is giving instructions to a computer so it can perform tasks for you:
  - Following a recipe to bake a cake
  - Following directions to get to a new place

- Computers need clear, **step by step** instructions.

What is programming?

- We use programming languages to communicate instructions to the computer (R, Python, Java, C...).

- The instructions are your "**code**".

What is programming?

- We use programming languages to communicate instructions to the computer (R, Python, Java, C...)

- The instructions are your “**code**”.

  Each line of code, is one step in your instructions:

  If you want the computer to add two numbers, you write a code that says: "add 2 and 3".

  If you want it to organize your list of favourite songs, you'd write code to sort the list.

The **R** language

- When you download and install R, you install the **“R interpreter”**

- The **“R interpreter”** reads and executes your R code one line at a time, so the computer understands your instructions.

The **R** language

- When you download and install R, you install the **“R interpreter”**
- The **“R interpreter”** reads and executes your R code one line at a time, so the computer understands your instructions.
- The **R interpreter** acts as a middleman:
  - It converts your R code into **“machine understandable”** language.
  - These instructions are then executed by the computer’s processor.

The **R** language

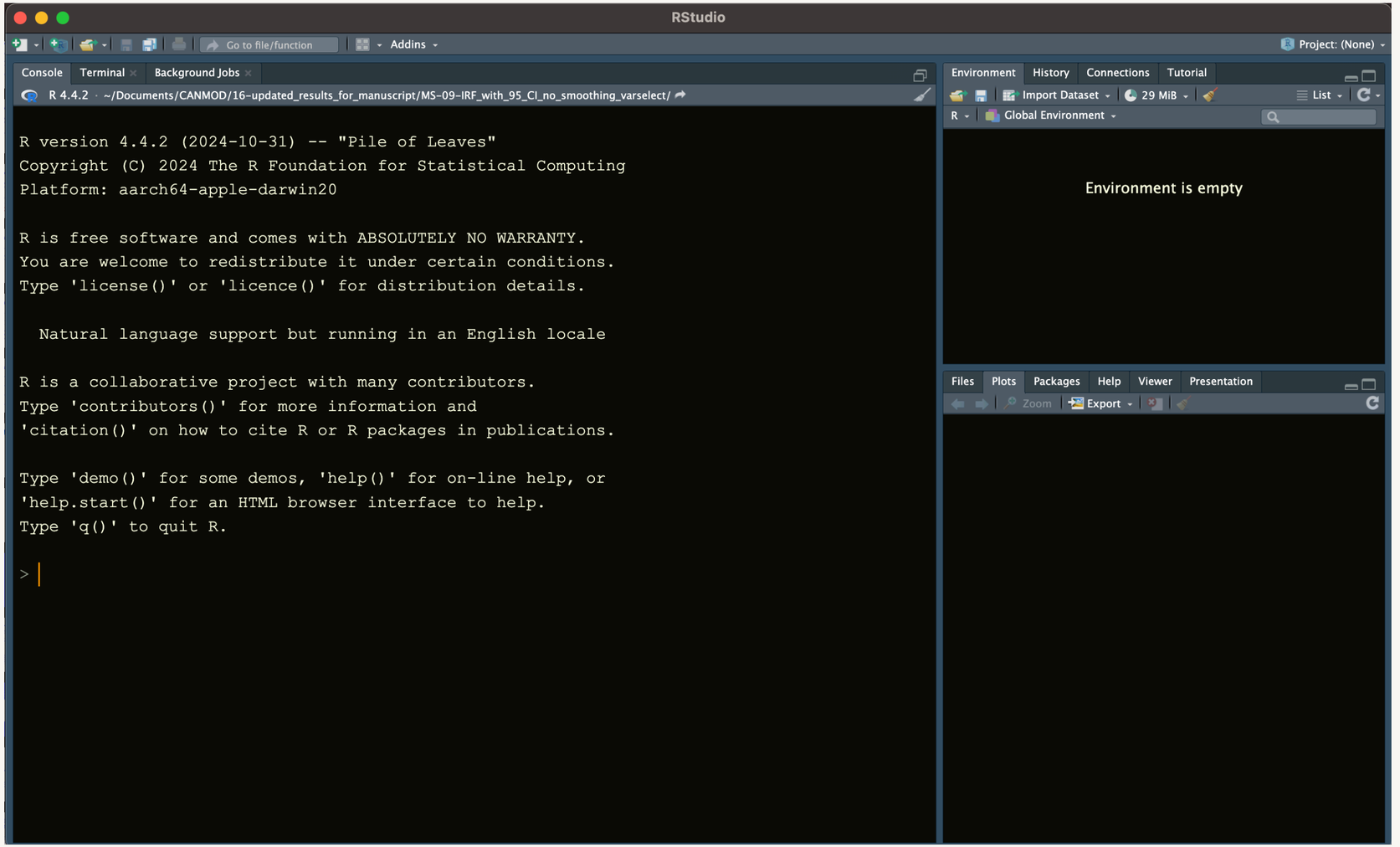
The interpreter reads the first line of code

The interpreter translates the line into machine instructions

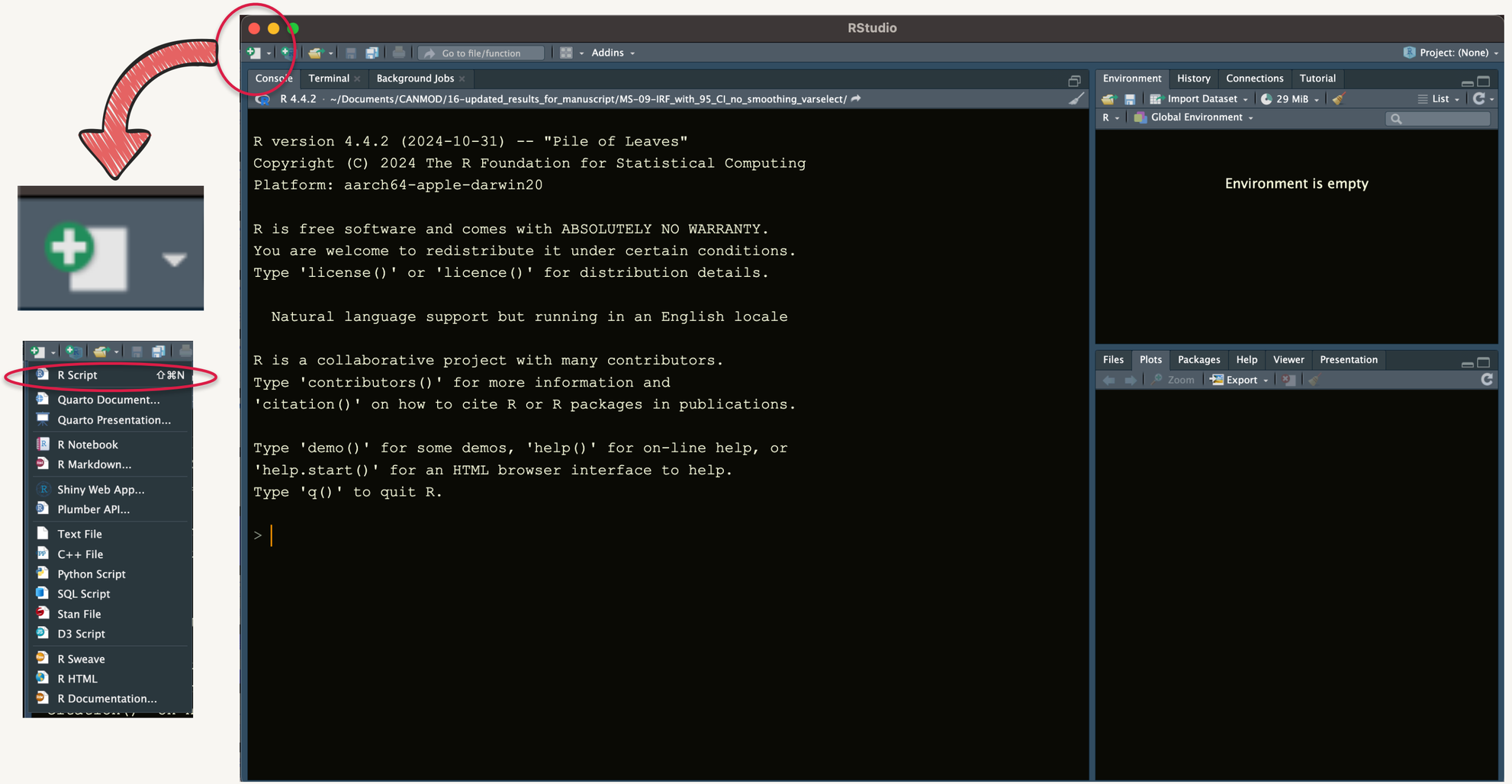
The computer executes the translated instruction.

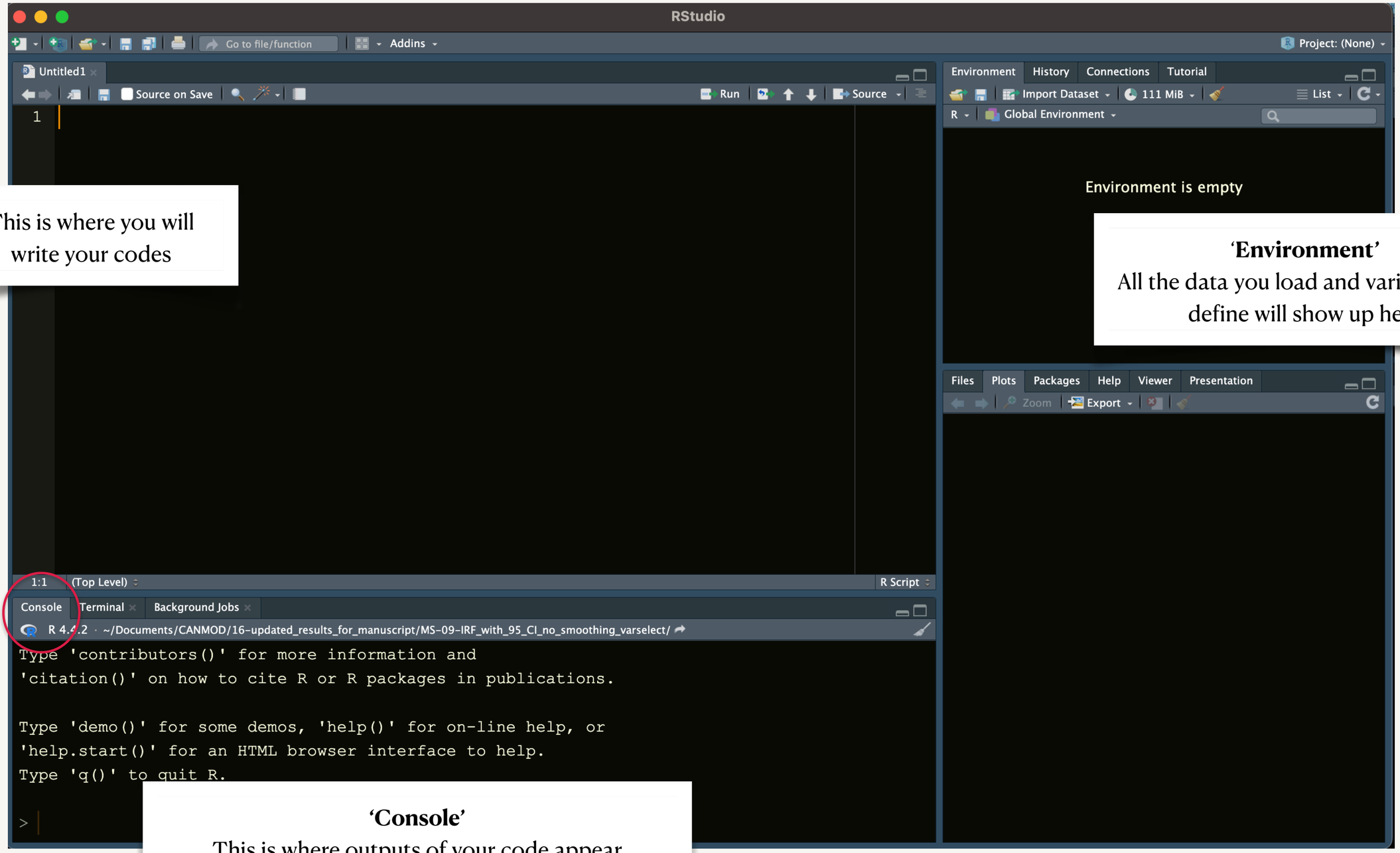
The interpreter translates and executes next line of code

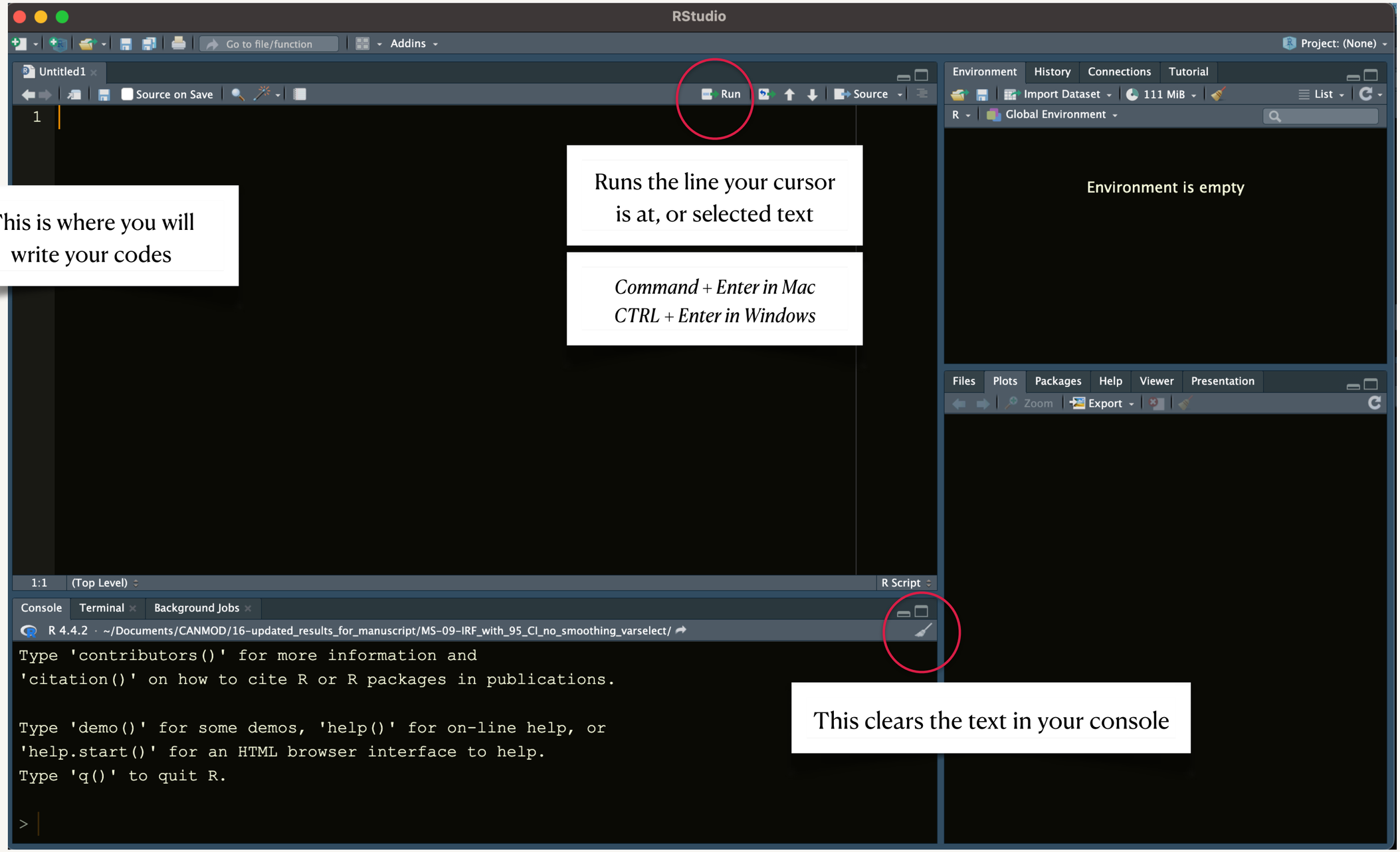
What is RStudio® and why use it?



What is RStudio® and why use it?




RStudio
Go to file/function
Addins
Project: (None)
Untitled1
Source on Save
Run
Source
1
Environment
History
Connections
Tutorial
Import Dataset
111 MiB
List
R
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
Presentation
Zoom
Export
1:1
(Top Level)
R Script
Console
Terminal
Background Jobs
R 4.4.2 · ~/Documents/CANMOD/16-updated_results_for_manuscript/MS-09-IRF_with_95_CI_no_smoothing_varselect/
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
This is where you will write your codes
'Environment'
All the data you load and variables you define will show up here
'Console'
This is where outputs of your code appear
It also allows you to run single lines of code at a time


RStudio
Go to file/function
Addins
Project: (None)
Untitled1
Source on Save
Run
Source
1
Environment
History
Connections
Tutorial
Import Dataset
111 MiB
List
R
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
Presentation
Zoom
Export
1:1
(Top Level)
R Script
Console
Terminal
Background Jobs
R 4.4.2 · ~/Documents/CANMOD/16-updated_results_for_manuscript/MS-09-IRF_with_95_CI_no_smoothing_varselect/
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
This is where you will write your codes
Runs the line your cursor is at, or selected text
Command + Enter in Mac
CTRL + Enter in Windows
This clears the text in your console

Run your first line of code!

- Type and run the following command in RStudio, and see the output in the console:

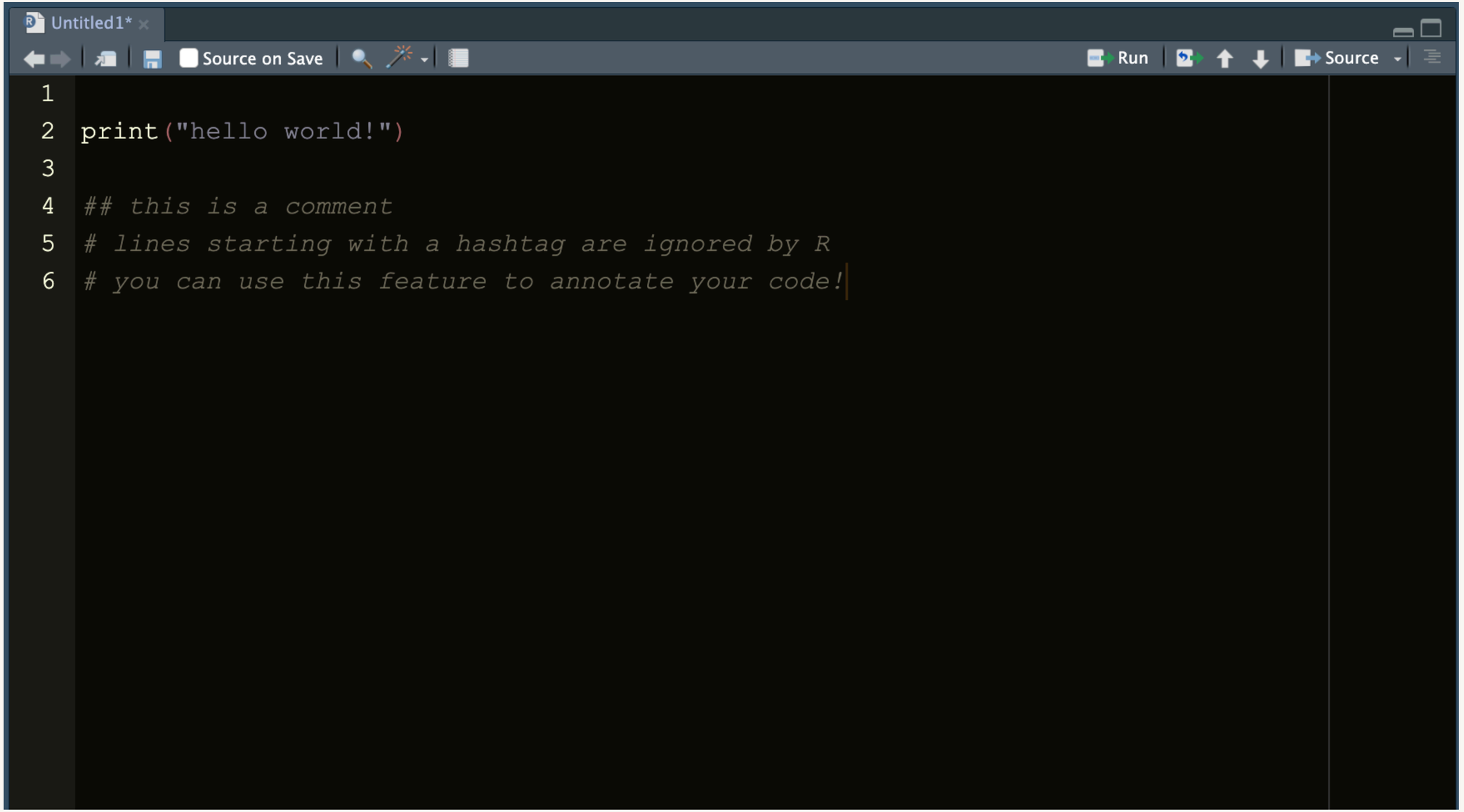
```
print(“hello world!”)
print(“This is my first code ever")
```

- `print()` will display whatever you put between double quotations.
- Some important notes:
  - Programming languages are case-sensitive: upper/lower case matters: `Print` is not the same as `print`
  - Write one command per line (you could separate commands with ; but for clarity stick to one per line)

```
print(“hello world!”); print(“this is a new command”)
print(“This is my first code ever")
```

- Try running this on the console.

Commenting on code:



 Save your first script!

- File > Save as > choose your filename and file destination

- Similar to any other file:
    - Command + S in Mac
    - Control + S in Windows

Building blocks of programming

Building block	Description	Examples
The ‘what’	**Data types and data structures**	A number, a sequence of numbers, a table, a name, a word, a sentence etc.
The ‘storage’	**Variables**	You load a table, and name it `my_table`. You now refer to that table with this name, whenever you want to perform actions on it.
The ‘how’	**Operators and functions**	Summation, a statistical test, a comparison between two values, removing a row in the table etc.
The ‘decisions’	**Control structures**	Checking certain conditions before performing certain tasks, repeating certain tasks for the same or different scenarios, etc.
The ‘toolbox’	**Packages and prewritten functions for R**	*‘ggplot’* for data visualization, *‘lmtest’* for performing statistical tests

Building blocks of programming

Building block	Description	Examples
The ‘what’	**Data types and data structures**	A number, a sequence of numbers, a table, a name, a word, a sentence etc.
The ‘storage’	**Variables**	You load a table, and name it `my_table`. You now refer to that table with this name, whenever you want to perform actions on it.
The ‘how’	**Operators and functions**	Summation, a statistical test, a comparison between two values, removing a row in the table etc.
The ‘decisions’	**Control structures**	Checking certain conditions before performing certain tasks, repeating certain tasks for the same or different scenarios, etc.
The ‘toolbox’	**Packages and prewritten functions for R**	*‘ggplot’* for data visualization, *‘lmtest’* for performing statistical tests

Building blocks of programming

Building block	Description	Examples
The ‘what’	**Data types and data structures**	A number, a sequence of numbers, a table, a name, a word, a sentence etc.
The ‘storage’	**Variables**	You load a table, and name it `my_table`. You now refer to that table with this name, whenever you want to perform actions on it.
The ‘how’	**Operators and functions**	Summation, a statistical test, a comparison between two values, removing a row in the table etc.
The ‘decisions’	**Control structures**	Checking certain conditions before performing certain tasks, repeating certain tasks for the same or different scenarios, etc.
The ‘toolbox’	**Packages and prewritten functions for R**	*‘ggplot’* for data visualization, *‘lmtest’* for performing statistical tests

Building blocks of programming

Building block	Description	Examples
The 'what'	**Data types and data structures**	A number, a sequence of numbers, a table, a name, a word, a sentence etc.
The 'storage'	**Variables**	You load a table, and name it `my_table`. You now refer to that table with this name, whenever you want to perform actions on it.
The 'how'	**Operators and functions**	Summation, a statistical test, a comparison between two values, removing a row in the table etc.
The 'decisions'	**Control structures**	Checking certain conditions before performing certain tasks, repeating certain tasks for the same or different scenarios, etc.
The 'toolbox'	**Packages and prewritten functions for R**	*'ggplot'* for data visualization, *'lmtest'* for performing statistical tests

Building blocks of programming

Building block	Description	Examples
The ‘what’	**Data types and data structures**	A number, a sequence of numbers, a table, a name, a word, a sentence etc.
The ‘storage’	**Variables**	You load a table, and name it `my_table`. You now refer to that table with this name, whenever you want to perform actions on it.
The ‘how’	**Operators and functions**	Summation, a statistical test, a comparison between two values, removing a row in the table etc.
The ‘decisions’	**Control structures**	Checking certain conditions before performing certain tasks, repeating certain tasks for the same or different scenarios, etc.
The ‘toolbox’	**Packages and prewritten functions for R**	*‘ggplot’* for data visualization, *‘lmtest’* for performing statistical tests

The ‘what’s: data types

Data Type	Description	Example

Numeric	Numbers (with or without decimals)	`2`
		`3.7`
		`245`

Integer	Whole numbers (declared with L)	`5L` `200L`

Character	Text (**strings**), enclosed in quotes	`“Hello”`
		`“this is a string”`
		`“b”`
		`“B”`

Logical	Boolean values: TRUE or FALSE	`TRUE` `FALSE`

Data types: check data types in R

You can check data types in R using the function `class()`

```
class(2.3)
class(5L)
class("hello world")
class(TRUE)
```

```
class(5)
class("5")
```

The ‘what’s: data structures

Data structures are the **containers** that hold and organize **data types** in R.

Data Structure	What It Stores	Example
Vector	A sequence of concatenated values (all same type)	`c(1, 2, 3, 4)` `c("apple", “strawberry”, “papaya”)`
Matrix	A table with rows and columns (all same type)	`matrix(1:6, nrow = 2, ncol = 3)`
Data Frame	A **table** with different types of data in each column	`data.frame(name = "Alice", age = 25)` `data.frame(name = c(“Alice”, “Laura”), age = c(25, 27))`
List	A flexible container that can hold different types of data Not in a table format!	`list(name = "Bob", scores = c(90, 85), passed = TRUE)`
Factor	A special structure for categorical data (groups or labels)	`factor(c("low", "medium", "high"))`

Data structures: vectors

- Vectors are a concatenated sequence of a single type of data.
- They are defined in R using the function `c()`, separated by commas

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

c("one", "two", "three")

c(TRUE, TRUE, FALSE, FALSE)
```

- If we create a vector that has both numeric and character values, the numeric values will get converted to a character data type: `c(1, 2, "abc")`

Define a few vectors and see the output!

Data structures: matrix

- Matrices are tables of information, of the same type (only numbers, only characters)
- They are defined in R using the function `matrix()`

```
matrix(vector_of_data,
       nrow = number_of_rows,
       ncol = number_of_columns,
       byrow = TRUE)
```

- Number of rows and columns should match the number of data points you provide in '`vector_of_data`'
- '`by_row`' determines how the matrix is filled .

```
matrix(c(1,200,34,56,78,12),
       nrow = 3,
       ncol = 2,
       byrow = TRUE)
```

Data structures: matrix

Try to define these matrices yourself:

1	1	1
2	0	5
3	1	6

“Red”	“Blue”	“Yellow”
“Purple”	“Green”	“Blue”
“Pink”	“Blue”	“Orange”

Data structures: dataframes

- Probably the most commonly used data structure for data analysis purposes.
- It is a table of information, in which every column can have a different data type.
- Each column can only have one data type: each column a vector!

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	170	FALSE
Yasaman	165.3	TRUE
Laura	166	TRUE

Character Numeric Logical

Data structures: dataframes

- Probably the most commonly used data structure for data analysis purposes.
- It is a table of information, in which every column can have a different data type.
- Each column can only have one data type.

TABLE FORMAT: all columns have the same number of rows!

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	170	FALSE
Yasaman	165.3	TRUE
Laura	166	TRUE

Character Numeric Logical

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	**NA**	FALSE
Yasaman	165.3	TRUE
Laura	166	**NA**

Use NA if unknown!

Data structures: try and define this yourself!

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	170	FALSE
Yasaman	165.3	TRUE
Laura	166	TRUE

```
data.frame(column1 = values_for_column1,
           column_2 = values_for_column2,
           column_3 = values_for_column2)
```

```
data.frame(name = c("ruthie", "Geraldine", "Martha", "Yasaman", "Laura"),
           Height = c(155.5, 162, 170, 165.3, 166),
           Attendance = c(FALSE, TRUE, FALSE, TRUE, TRUE))
```

Data structures: lists

- Lists are versatile containers of data.
- A list can hold different types of data at the same time (numbers, text, vectors, data frames even other lists!)
- *Think of a list like a drawer that holds mixed items: in each drawer you can put any data structure you want*

```
list(name = "Alice", age = 25, scores = c(90, 85, 88))
```

- Very good for more complex data forms, such as *trees, or outputs of complex statistical tests.*

Data structures: factors

- Factors are structures that are used for **categorical variables.**
- A categorical variable can take on one of a limited, and usually fixed, number of possible values, based on some **qualitative** property.
- Usually useful in the context of statistical modelling. You can have categorical variables without converting them into a factor.
- Examples: yes/no , low,/medium/high

```
factor(c("small", "medium", "large", "medium", "small"))
```

```
> factor(c("small", "medium", "large", "medium", "small"))
[1] small  medium large  medium small
Levels: large medium small
```

The ‘storage’: variables

- When you define a data structure, you can store it in a space in the memory, and label it with a name.
- **Variables** are the identifier or the named space in the memory, in which you store your defined data structures.
- Data structures and can be **referenced** and **manipulated** later in the program.

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

attendees = c(“sana”, “Laura”, “someone”)

mytable = data.frame(name = c(“ruthie”, “Geraldine”, “Martha”, “Yasaman”, “Laura”),
           Height = c(155.5, 162, 170, 165.3, 166),
           Attendance = c(FALSE, TRUE, FALSE, TRUE, TRUE))
```

The ‘storage’: variables

- Names of the variables should follow certain rules:
  - Can only contain letters, numbers, underscores, and periods
  - Must start with a letter (can not start with number or signs)
  - Case sensitive: `myVector` is not the same as `myvector`
  - Symbols and white space (except “_”) NOT allowed: `'my vector'` is not accepted, neither is `my_ha$htag.`
  - Certain names are reserved: `TRUE = 1` will return an error: TRUE is reserved for logical data type.

Variables: try and define variables

- Variables are assigned in R using one of two notations:
  - ‘=‘ `X = 20`
  - ‘<-‘ `X <- 20`
- Both can be used, but they have differences in certain contexts.

```
A = 1
b = 2
mynumber = “657”

test_factor = factor(c(“small", "medium", "large", "medium", "small"))


my_first_matrix = matrix(c(1,200,34,56,78,12),
       nrow = 3,
       ncol = 2,
       by_row = TRUE)
```

Variables: check variable types

As discussed before, you can check variable types using the `class()` function.

```
class(test_factor)
```

```
class(my_first_matrix)
```

Break

Recap

- We have introduced the ‘What’ and the ‘Storage’.
- Data types in R are **numeric**, **character**, and **logical**.
  - **Numerics** are any number (whole or decimal)
  - **Characters or strings** are any combination of letters and symbols, confined within “”

    (character is usually referred to a single letter/symbol and string refers to multiple letters/word/sentence)
  - **Logicals** are either TRUE/FALSE. These are usually outcomes of a comparison or condition:

    Is `a` larger than `b`? Is there missing data?

Recap

- We have introduced the 'What' and the 'Storage'.
- Data types in R are **numeric**, **character**, and **logical**.
  - **Numerics** are any number (whole or decimal)
  - **Characters or strings** are any combination of letters and symbols, confined within " "

    (character is usually referred to a single letter/symbol and string refers to multiple letters/word/sentence)
  - **Logicals** are either TRUE/FALSE. These are usually outcomes of a comparison or condition:

    Is `a` larger than `b`? Is there missing data?
- There are several ways in which we can store data in R. The most commonly used ones are **vectors, matrices, and data frames**.
  - **Vectors** are a concatenated sequence of any data type (same type).
  - **Matrices** are a table of any data type (same data type).
  - **Dataframes** are a table that allows different data types.

Accessing data in data structures: vectors

The = sign is what assigns the vector to the 'variable berry'

Note that white space is allowed in your strings

```
berry = c("strawberry", "blueberry", "raspberry", "golden berry")
```

Accessing data in data structures: vectors

The = sign is what assigns the vector to the 'variable berry'

Note that white space is allowed in your strings

```
berry = c("strawberry", "blueberry", "raspberry", "golden berry")
```

```
berry[1]
```

```
berry[2]
```

Accessing data in data structures: vectors

The = sign is what assigns the vector to the 'variable berry'

Note that white space is allowed in your strings

```
berry = c("strawberry", "blueberry", "raspberry", "golden berry")
```

```
berry[1]

berry[2]
```

```
i = 1
berry[i]
```

Accessing data in data structures: vectors

The = sign is what assigns the vector to the 'variable berry'

Note that white space is allowed in your strings

```
berry = c("strawberry", "blueberry", "raspberry", "golden berry")
```

```
berry[1]

berry[2]
```

```
i = 1
berry[i]
```

```
berry[1:3]
```

That counts all numbers from 1 to 3

```
1:10

2:6
```

Accessing data in data structures: vectors

The = sign is what assigns the vector to the 'variable berry'

Note that white space is allowed in your strings

```
berry = c("strawberry", "blueberry", "raspberry", "golden berry")
```

```
berry[1]

berry[2]
```

```
i = 1
berry[i]
```

```
berry[1:3]
```

That counts all numbers from 1 to 3

```
berry[c(1,4)]
```

That returns a vector including 1 and 4

```
1:10

2:6
```

Accessing data in data structures: vectors

The = sign is what assigns the vector to the 'variable berry'

Note that white space is allowed in your strings

```
berry = c("strawberry", "blueberry", "raspberry", "golden berry")
```

```
berry[1]

berry[2]
```

```
i = 1
berry[i]
```

```
berry[1:3]
```

That counts all numbers from 1 to 3

```
berry[c(1,4)]
```

That returns a vector including 1 and 4

```
i = 2:4
berry[i]

i = c(1,4)
berry[i]
```

```
1:10

2:6
```

Accessing data in vectors

```
v = 1:100
v = c(1,2,3,4,…, 100)
```

Access the first and last number and store them in a variable named `t1`.

Access 50 and 100 at the same time and store them in a variable named `t2`.

Access the second half of the vector and store them in a variable named `t3`.

Access all numbers from 30 to 66 and store them in a variable named `t4`.

Accessing data in vectors

```
v = 1:100
v = c(1,2,3,4,…, 100)
```

Access the first and last number and store them in a variable named `t1`.

```
t1 = v[c(1, 100)]
```

Access 50 and 100 at the same time and store them in a variable named `t2`.

```
t2 = v[c(50, 100)]
```

Access the second half of the vector and store them in a variable named `t3`.

```
t3 = v[50:100]
```

Access all numbers from 30 to 66 and store them in a variable named `t4`.

```
t4 = v[30:66]
```

Accessing data in data structures: matrix

- Matrices are two-dimensional data structures.
- 3 rows and 4 columns.

```
M[i,j]
```

Row number(s) of the element(s) you want to access

Column number(s) of the element(s) you want to access

$$M = \begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

Row 1

Row 2

Row 3

column 1

column 2

column 3

column 4

Accessing data in data structures: matrix

- Matrices are two-dimensional data structures.
- 3 rows and 4 columns.

`M[i,j]`

Row number(s) of the element(s) you want to access

Column number(s) of the element(s) you want to access

$$M = \begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

Row 1
Row 2
Row 3

column 1
column 2
column 3
column 4

- `i` and `j` can be any number or sequence of numbers within the range of the number of rows and columns in M.
- A convenient way of accessing entire rows or columns is only providing `i` or `j`:

`M[i,]` all of i'th row(s)

`M[,j]` all of j'th column(s)

Accessing data in matrices

```
M =
```
$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

1 `M[2, 4]`

$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & \boxed{8} \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

2 `M[1:2, 2:3]`

$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

3
```
i = 2
j = 2:4
M[i, j]
```
$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

4 `M[2,]`

$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

5
```
i = c(1,3)
M[i,]
```
$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

6
```
j = c(1,4)
M[,j]
```
$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

Accessing data in matrices

```
M =
```
$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

1 `M[2, 4]`

$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & 6 & 1 & \boxed{8} \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

2 `M[1:2, 2:3]`

$$\begin{pmatrix} 7 & \boxed{3} & \boxed{9} & 2 \\ 4 & \boxed{6} & \boxed{1} & 8 \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

3
```
i = 2
j = 2:4
M[i, j]
```

$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ 4 & \boxed{6} & \boxed{1} & \boxed{8} \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

4 `M[2,]`

$$\begin{pmatrix} 7 & 3 & 9 & 2 \\ \boxed{4} & \boxed{6} & \boxed{1} & \boxed{8} \\ 5 & 10 & 3 & 6 \end{pmatrix}$$

5
```
i = c(1,3)
M[i,]
```

$$\begin{pmatrix} \boxed{7} & \boxed{3} & \boxed{9} & \boxed{2} \\ 4 & 6 & 1 & 8 \\ \boxed{5} & \boxed{10} & \boxed{3} & \boxed{6} \end{pmatrix}$$

6
```
j = c(1,4)
M[,j]
```

$$\begin{pmatrix} \boxed{7} & 3 & 9 & \boxed{2} \\ \boxed{4} & 6 & 1 & \boxed{8} \\ \boxed{5} & 10 & 3 & \boxed{6} \end{pmatrix}$$

3x2 matrix

Accessing data in data structures: data frames

- Data frames are two-dimensional data structures.
- Let's say we have this data table:

```
mytable = data.frame(name = c("ruthie", "Geraldine", "Martha", "Yasaman", "Laura"),
                     Height = c(155.5, 162, 170, 165.3, 166),
                     Attendance = c(FALSE, TRUE, FALSE, TRUE, TRUE))
```

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	170	FALSE
Yasaman	165.3	TRUE
Laura	166	TRUE

- Accessing data is similar to matrices, but slightly more versatile
  - All methods mentioned before still work:

```
mytable[i, j]

mytable[i,]

mytable[,j]
```

- Where `i` and `j` can be any sequence of numbers within the dimensions of this table.

Accessing data in data structures: data frames

Additional ways in which you can access data in data frames:

1. Using row/column names

```
mytable[i, “name”]
mytable[, “Height”]
mytable[“row_name”,]
```

2.Using the $ sign to access columns

```
mytable$name
mytable$Height[i]
```

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	170	FALSE
Yasaman	165.3	TRUE
Laura	166	TRUE

Accessing data in data frames

```
mytable = data.frame(name = c(“ruthie”, “Geraldine”, “Martha”,
“Yasaman”, “Laura”),
                     Height = c(155.5, 162, 170, 165.3, 166),
                     Attendance = c(FALSE, TRUE, FALSE, TRUE, TRUE))
```

```
mytable$height

mytable$name


mytable[c(1, 3), “name”]
```

name	Height	Attended workshop
Ruthie	155.5	FALSE
Geraldine	162	TRUE
Martha	170	FALSE
Yasaman	165.3	TRUE
Laura	166	TRUE

The ‘how’: operators and functions

- Operators and functions facilitate the **actions** you perform on your data.

- Examples:
    - Summing up a column in a table
    - Finding the minimum value in a column of your data
    - Comparing the length of one vector with another
    - Sorting data
    - Loading data into your R environment
    - Etc..

The ‘how’: Functions

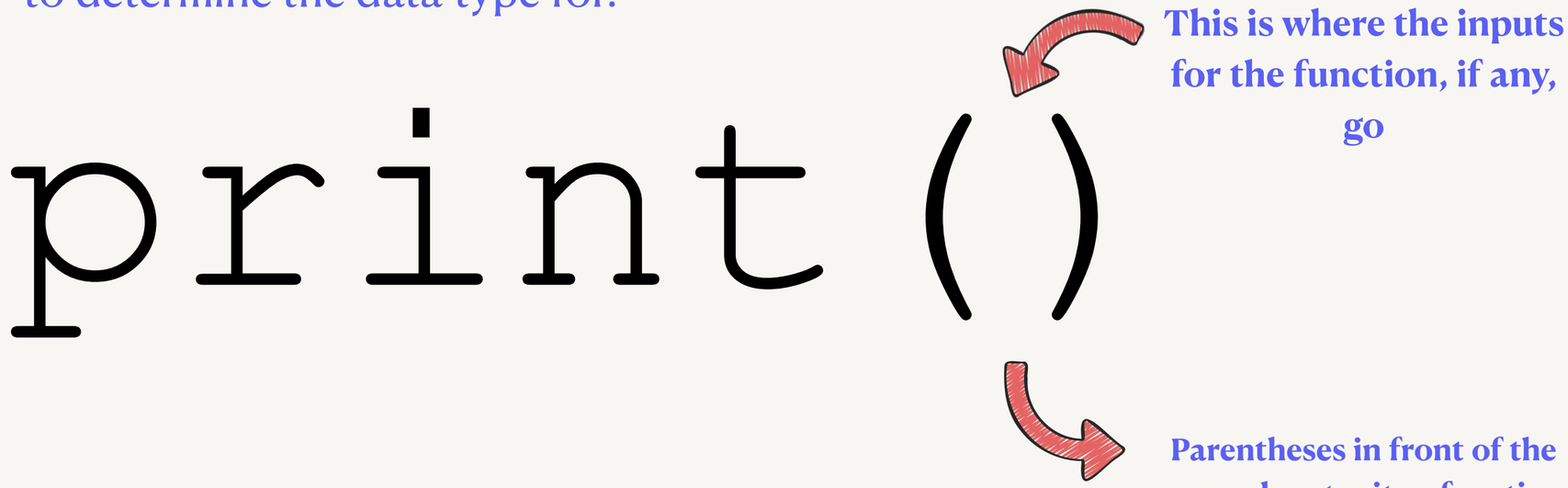
- Functions are self-contained **blocks of code** that perform specific tasks.
- When you run a function, you ‘**call**’ it by its ‘*name*’. The **block of code contained within that function** is executed to perform tasks defined within it.

Pre-defined

- Functions may or may not need additional information input from you.

The ‘how’: Functions

- Example: `print(), class()`
- `print()` and `class()` are predefined functions in R.
- When you call it, its predefined ‘source code’ is executed.
- `print()` needs an input from you: what to print? `class()` also needs an input: what is it that you want to determine the data type for?



The ‘how’: Functions

`print()`

- Whatever you input in the parenthesis for `print()`, it will output.

- It can be a variable with a numeric or character type, or directly a character (confined in double-quotes)

```
A = 2
print(A)

name = “ARC”
print(name)

print(“hello world!”)
```

Functions: some basic functions to try!

You can get the documentation on how to use a function by searching it online, or simply running `?'name_of_function'` in R.

`ls()`	Lists all the variables defined or loaded into your environment.	Just run `ls()` and see!

`sum()`	Inputs a vector of numbers, and returns the sum of all values.	`my_vector = c(1, 2, 6, 9)`
		`sum(my_vector)`
		`sum_of_my_vector = sum(my_vector)`

`max()`
`min()`
	Inputs a vector of numerics, and returns the maximum/minimum value.	`my_vector = c(1, 2, 6, 9)`
		`max(my_vector)`
		`min(my_vector)`

`length()`	Inputs a vector and returns the length of it (the number of elements in in).	`letters = c("b", "g", "t", "m")`
		`length(letters)`
		`length(my_vector)`

`nrow()` `ncol()`	Inputs a matrix or a data frame, and returns the number of rows/columns	`nrow(my_matrix)` `ncol(my_matrix)`
`colnames()` `rownames()`	Inputs a matrix or data frame, and returns the names of columns/rows. It also allows you to assign or change column/row names.	`rownames(my_matrix)` `colnames(my_dataframe)`

The ‘how’: What are operators?

In programming languages, operators are **symbols** that perform specific operations on values or variables.

Arithmetic operator

Used to sum values

```
a + b
```

Sums `a` and `b`

The ‘how’: What are operators?

In programming languages, operators are **symbols** that perform specific operations on values or variables.

Arithmetic operator

Used to sum values

`a + b`

Sums `a` and `b`

Relational operator

Used to compare values

`a > b`

Tells us if `a` is larger than `b`

The ‘how’: What are operators?

In programming languages, operators are **symbols** that perform specific operations on values or variables.

Arithmetic operator

Used to sum values

```
a + b
```

Sums `a` and `b`

Relational operator

Used to compare values

```
a > b
```

Tells us if `a` is larger than `b`

Assignment operators

Used to assign values to a variable

```
a = 3
b = a
```

Assigns the value of a to 3

Assigns the value of b to whatever a is

The ‘how’: What are operators?

1. Arithmetic operators

Operator	Description
`a + b`	Sums two variables
`a - b`	Subtracts two variables
`a * b`	Multiply two variables
`a / b`	Divide two variables
`a ^ b`	Exponentiation of a variable
`a %% b`	The remainder of a variable
`a %/% b`	Integer division of variables

The ‘how’: What are operators?

1. Arithmetic operators

Operator	Description
`a + b`	Sums two variables
`a - b`	Subtracts two variables
`a * b`	Multiply two variables
`a / b`	Divide two variables
`a ^ b`	Exponentiation of a variable
`a %% b`	The remainder of a variable
`a %/% b`	Integer division of variables

- They are performed on **numeric** values and variables, and the outputs are **numeric**.
- You can store the output of these operators in a new variable (numeric)

```
a = 1
b = 2
sum_val = a + b
sum_val
class(sum_val)
```

The ‘how’: What are operators?

2. Relational operators

Operator	Description
`a > b`	Tests for greater than
`a < b`	Tests for smaller than
`a >= b`	Tests for greater or equal than
`a <= b`	Tests for smaller or equal than
`a == b`	Tests for equality
`a != b`	Tests for inequality

- These operators are used for comparisons.
- **The result of the comparison is a TRUE/FALSE (logical)**
- These are useful in **condition structures.**

  If `a` larger than `b`, then perform task 1.

  If `a` smaller than `b`, then perform task 2.

  If `a` equal to `b`, then perform task 3.

The ‘how’: What are operators?

2. Relational operators

Operator	Description
`a > b`	Tests for greater than
`a < b`	Tests for smaller than
`a >= b`	Tests for greater or equal than
`a <= b`	Tests for smaller or equal than
`a == b`	Tests for equality
`a != b`	Tests for inequality

- These operators are used for comparisons.
- **The result of the comparison is a TRUE/FALSE (logical)**
- These are useful in **condition structures.**

If `a` larger than `b`, then perform task 1.

If `a` smaller than `b`, then perform task 2.

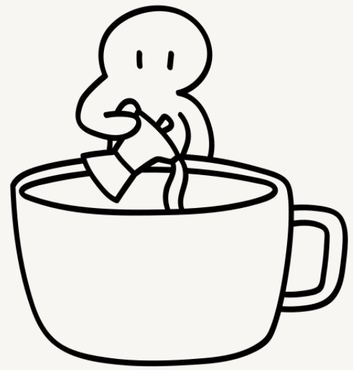
If `a` equal to `b`, then perform task 3.

These two operators can be performed on any data type

‘=‘ and ‘==‘ are not the same!

‘=’ is the “assignment” operator, used to assign values/data to a variable.

Whatever is on the right side, is poured into the container defined on the left.

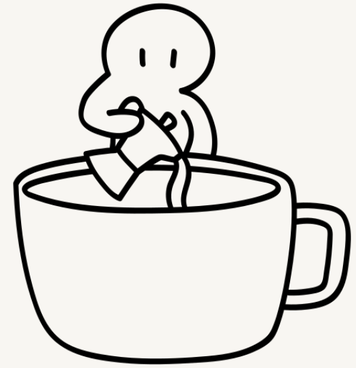


```
name_vector = mytable$names

small_matrix = my_matrix[2:3, 1:2]

comparison = (a > b)
```

‘=‘ and ‘==‘ are not the same!



- ‘=’ is the “assignment” operator, used to assign values/data to a variable.

  Whatever is on the right side, is poured into the container defined on the left.

```
name_vector = mytable$names
small_matrix = my_matrix[2:3, 1:2]
comparison = (a > b)
```

- ‘==’ is a “relational” operator, used to compare what’s on the right with what’s on the left. The result of this comparison is always a logical: either TRUE or FALSE.

```
a == b
mytable$Height[1] == mytable$Height[2]
```

Practice (for home)

Play around with the “==“ operator and see what it looks like to compare different data types and structures on both sides. For example:

```
a == b
```

What happens if `a` is a number and `b` is a vector of numbers?

What happens if `a` is a character and `b` is a number?

What happens if `a` is a vector of numbers and `b` is a matrix?

What if one side is a data frame? Is the comparison valid?

...

Practice

Fruit	Weight	Price_per_kg
Apple	150	3.5
Banana	120	1.2
Cherry	10	10.0

1. Create the data frame shown and call it ‘`fruits`’.

2. Print the data frame to look at its contents.

3. Access the ‘`Fruit`’ column and store it in a variable called ‘fruit_vector’

4. Retrieve the weight of the second fruit (Banana). Store this in a variable called ‘`banana_weight`’

5. Print only the first two rows from the data frame.

6. Create a new data frame called ‘`fruit_prices`’ that contains only the "`Fruit`" and "`Price_per_kg`" columns, and store it in a variable called ‘`small_df`’.

Practice

7. Convert the weight of each fruit from grams to kilograms by dividing by 1000. Multiply the converted weight by the price per kilogram to get the total price. Store the result in a variable called `total_price`.

8. Calculate the average weight of the fruits using the `mean()` function and store it in mean_weight.

9. Find the highest price per kilogram using the `max()` function and store it in max_price.

10. Get the column names for the ‘fruits’ data frame using the `colnames()` function.

11. Get the number of rows and columns in the data frame using `nrow()` and `ncol()` functions.

Break

The ‘decisions’: control structures

- Control structures allow you to make decisions and repeat tasks.
- Generally speaking, control structures can be categorized into::
  - **Conditions**

    if “condition holds” then “perform task(s)”
  - **Loops**

    Sometimes you want to repeat similar tasks on different data. Loops automate this process.

Conditions

Open bracket always on the same line as if()

```
if(condition) {

  Task 1
  Task 2
  …

  }
```

Conditions

Anything that is type logical can go here.

Open bracket always on the same line as if()

```
if(condition) {

  Task 1
  Task 2
  …

  }
```

Conditions

Anything that is type logical can go here.

```
if(condition) {

  Task 1
  Task 2
  …

  }
```

```
a > b
a == b
fruits$Weight[1] > 100
class(2) == “numeric”
```

If the value of what goes in the parenthesis is TRUE, then whatever command is enclosed in the {} will be executed.

Conditions

```
a = 23
b = 50

if(a > b){

  print(“a is larger”)

}


if(a < b){

  print(“b is larger”)

}
```

If a > b returns ‘TRUE’, then the commands between the brackets will run.
If a > b returns ‘FALSE’, then everything between the brackets will be skipped.

Conditions

Let's say we have two variables a, b, with a value of 10, and 20, respectively. Write a code using condition structures that prints the bigger number.

```
a = 10
b = 20

if(a > b){
  print(a)
}

if(b > a){

   print(b)
}
```

Conditions

Let's say we have two variables a, b, with a value of 10, and 20, respectively. Write a code using condition structures that prints the bigger number.

```
a = 10
b = 20

if(a > b){
  print(a)
}

if(b > a){

   print(b)
}
```

```
a = 10
b = 20

if(a > b){
  print(a)
} else {

  print(b)

}
```

Conditions

Recall the **relational operators**, and how they are used for comparisons.

Operator	Description
`a > b`	Tests for greater than
`a < b`	Tests for smaller than
`a >= b`	Tests for greater or equal than
`a <= b`	Tests for smaller or equal than
`a == b`	Tests for equality
`a != b`	Tests for inequality

Conditions

Recall the **relational operators**, and how they are used for comparisons.

Operator	Description
`a > b`	Tests for greater than
`a < b`	Tests for smaller than
`a >= b`	Tests for greater or equal than
`a <= b`	Tests for smaller or equal than
`a == b`	Tests for equality
`a != b`	Tests for inequality

Logical operators

Operator	Description
`!`	Logical NOT
`&&`	Logical AND
`||`	Logical OR

Conditions: logical operators practice

```
a = 1, b = 2, c = 3, d = 4
```

Operation	Outcome
`a > b`	
`a == a`	
`(a < b) && (a < c)`	
`(a > b) && (a < c)`	
`(a == b) || (a < d)`	
`!(a < b)`	
`class(a) == “numeric”`	
`(a+b)!= c`	

Conditions: logical operators practice

`a = 1, b = 2, c = 3, d = 4`

Operation	Outcome
`a > b`	FALSE
`a == a`	TRUE
`(a < b) && (a < c)`	TRUE
`(a > b) && (a < c)`	FALSE
`(a == b) || (a < d)`	TRUE
`!(a < b)`	FALSE
`class(a) == “numeric”`	TRUE
`(a+b)!= c`	FALSE

Practice 1

Using if (or if-else) statements, write a code that checks if a number is positive, negative, or zero and prints the result.

```
number = -5
```

Practice 1

Using if (or if-else) statements, write a code that checks if a number is positive, negative, or zero and prints the result.

```
number = -5

if(number == 0){
   print(“number is zero”)
}

if(number > 0){
   print(“number is positive”)
}

if(number < 0){
   print(“number is negative”)
}
```

There are more efficient ways to write this code using else-if statements.

Practice 2

Write an if statement that checks if a person is 18 or older and prints "You can vote!". Otherwise, print "You cannot vote yet."

```
age = 17
```

Practice 2

Write an if statement that checks if a person is 18 or older and prints "You can vote!". Otherwise, print "You cannot vote yet."

```
age = 17

if(age >= 18){
   print(“You can vote!”)
}

if(age < 18){
   print(“You cannot vote yet”)
}
```

Loops

Loops are used in programming to **repeat a set of instructions multiple times** without writing the same code again and again.

Imagine you need to print numbers 1 to 5. Without a loop, you would have to write:

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Loops

Loops are used in programming to **repeat a set of instructions multiple times** without writing the same code again and again.

Imagine you need to print numbers 1 to 5. With a loop, you would instead write:

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

This goes through numbers 1 to 5 (identified by the variable `i`) and print the variable in each iteration.

Loops: How to write a loop?

```
for (variable in sequence) {
   Task1
   Task2
   Task3
}
```

Variable → A variable that stores the current value of the loop in the sequence.
Sequence → A set of values the loop goes through (e.g., numbers 1 to 5, names of fruits, etc).
Loop body {Task1, Task2, Task3} → The code inside the { } runs once for each value in the sequence.

Loops: example

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

i starts at one, and increments through all values defined in the sequence.
At every value of I, the tasks defined within the brackets are executed.

```
for (i in c("banana", "apple", "berry")) {
  print(i)
}
```

```
for (i in fruits$Fruit) {
  print(i)
}
```

The ‘toolbox’: Functions and packages

- When you download and install R, several pre-written functions and packages of functions come with it.

- The basic package of functions in R is called ‘`base`’: this incorporates most of what you need for basic programming in R.

- There are usually packages for basic statistical testing and plotting included too. For more complex tasks, you will have to find the package you need, and download it from **CRAN**.

- See a list of available packages on CRAN:
  https://cran.r-project.org/web/packages/available_packages_by_name.html

The ‘toolbox’: Installing additional packages

- In R, you can install packages using the `install.packages()` function:

```
install.packages(“package name in quotations”)
install.packages(“ggplot2”)
install.packages(“lme4”)
```

- Once installed, you load desired packages by running:

```
library(name_of_package_no_quotes)
library(ggplot2)
library(lme4)
```

- You can find detailed documentation of all packages on CRAN online.

- As mentioned before, you can always get instructions for packages and functions within them by running:

```
?print
?ggplot2
```

Final notes and prep for tomorrow:

- Familiarize yourself with the RStudio environment (creating and saving new scripts, writing code and commenting).

- Try to get comfortable with typing in commands, and running them in RStudio.

- Play around with variable types and operators: don't be afraid to try things!

- If possible, try to write yourself a few loops and conditions. **Do not copy and paste! Typing in blocks of code helps you learn and remember.**

- Almost any programming related information can be found online these days! Don't be shy to google your questions.

To do's for tomorrow:

- Install the packages `'ggplot2'` and `'stringr'` using the function `install.packahes()`.

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

- Download the file provided on the workshop's GitHub and have it ready for tomorrow.

https://github.com/lcamelo10/iWOMEN-in-ARC/tree/main/Workshops_material/Introduction%20to%20programming%20in%20R