BASIC R

APP-BIOINFO-2024 R SERIES

Duy Dao
khuongduying@gmail.com
2024, Nov 23th

Content

- Overview

- Install R

- Basic R

- Data types and Structures

- R Functions

Overview

What is R?

- A Programming/Statistical Language

- A powerful language for statistical computing and data analysis.

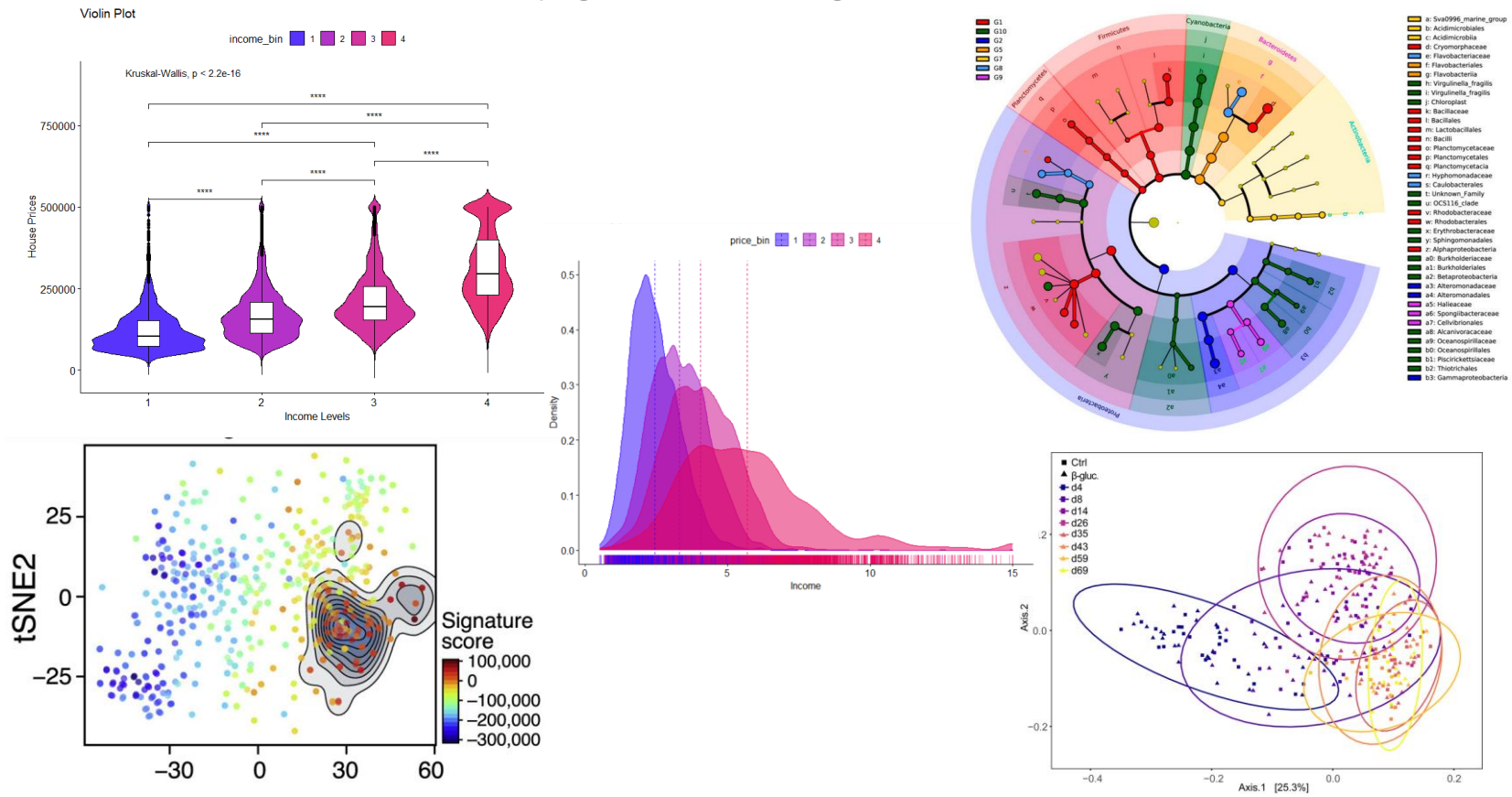- Widely used in bioinformatics and life sciences.

Why Use R for Bioinformatics Analysis?

1. Comprehensive Statistical Tools

2. Rich Ecosystem of Bioinformatics Packages

3. Data Visualization

4. Handling High-Dimensional Data

5. Open Source and Active Community

6. Reproducibility

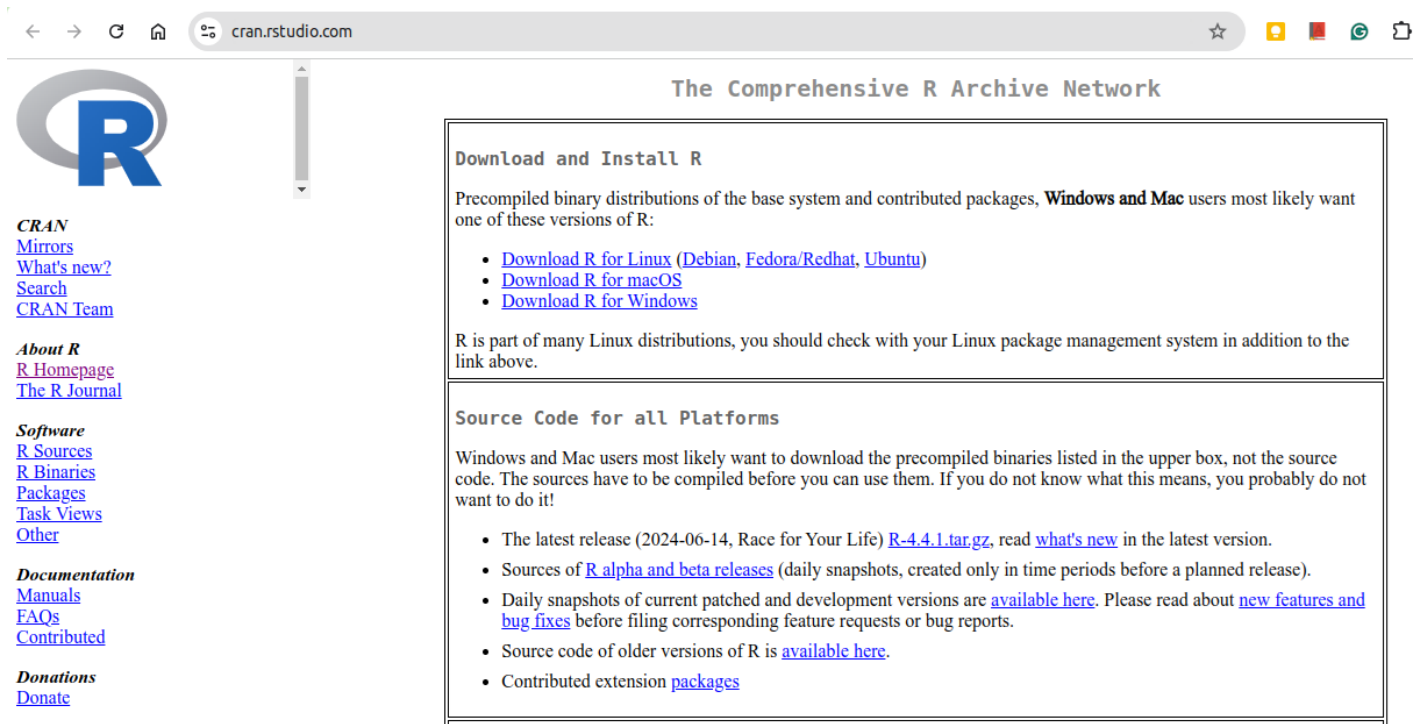7. Free

Why Use R for Bioinformatics Analysis?

Generation of publication-quality graphs and figures.

Install R

Install R

Install R by accessing to this R's original website

https://cran.rstudio.com/

R and RStudio

R Studio IDE (Integrated development environment - IDE)

Install: https://posit.co/download/rstudio-desktop/

- Enhances productivity with features like syntax highlighting, debugging tools, and version control.
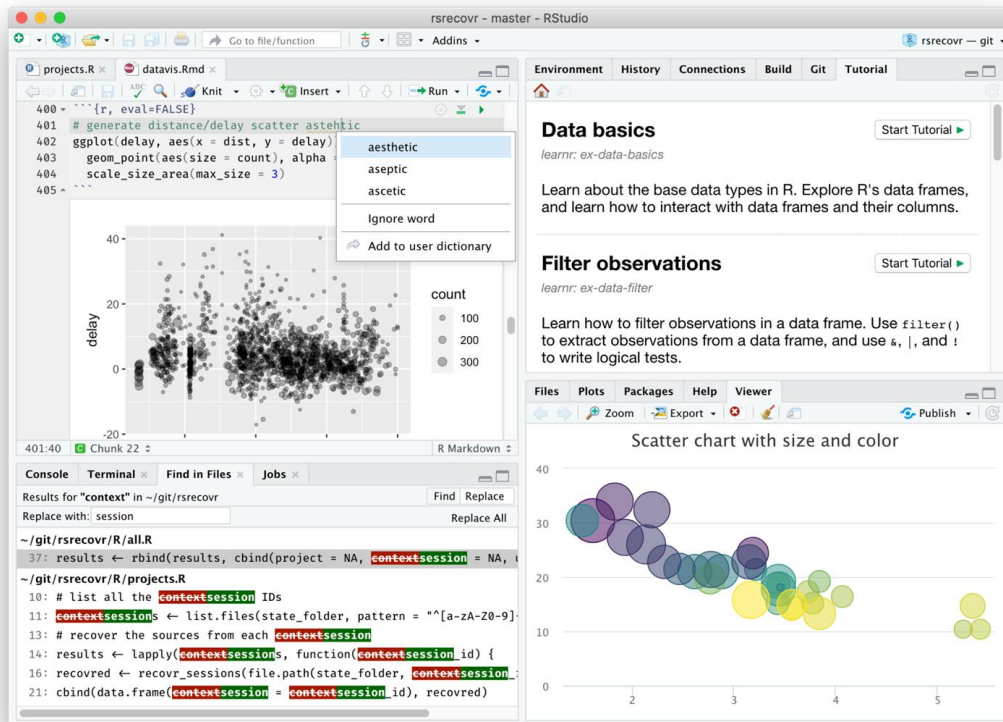
Icon for R

Icon for RStudio

R and RStudio

Alternative IDE: VScode

Install: https://code.visualstudio.com/
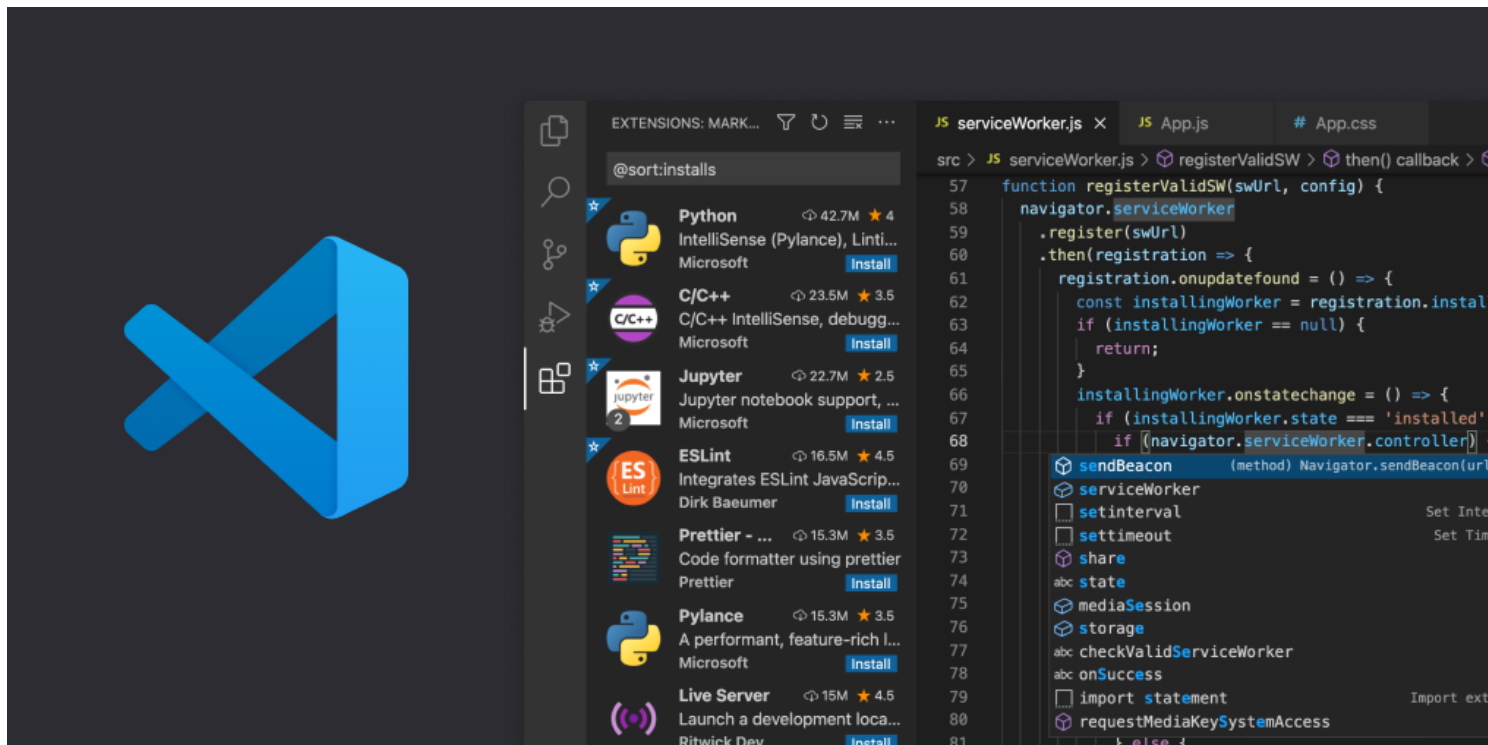
First R

The R script

```r
# Import library
library(ggpubr)
library(tidyverse)
library(plyr)

# Load data
data("ToothGrowth")
print(ToothGrowth)

# Error plots
# ggerrorplot()
# Mean +/- standard deviation
ggerrorplot(ToothGrowth,
            x = "dose", y = "len",
            desc_stat = "mean_sd")
```

Comments for each code block

Import libraries

Load data

Code to visualize the data

11

Working directories

A basic R project set up

Your Project Name
- .Rproj (R Project File)
- Data
- Script
  - Functions
  - Analysis A
  - Analysis B
- Output
  - Plots
  - Data

https://martinctc.github.io
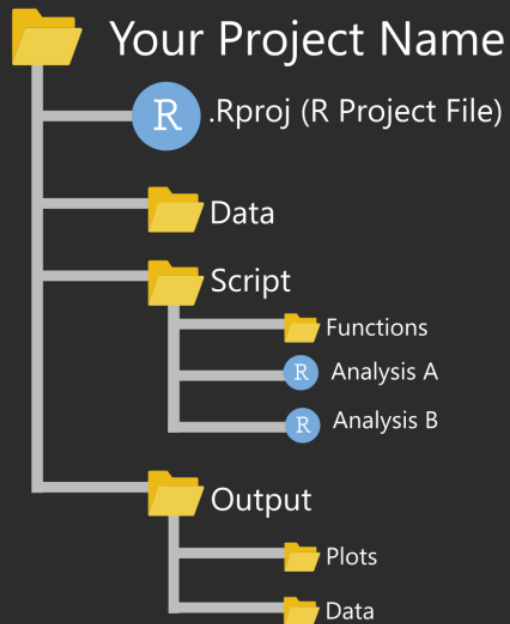
Working Directory
- The default folder where R looks for files to read and saves outputs.
- It's essential to know and set your working directory for organized data analysis.

```r
# First, have a look at the
current working directory
getwd()
# Change to your desired
directory
setwd()
# List the file in the directory
dir()
```

Installing and Loading R Packages

R packages are extensions to the R statistical programming language.

R packages contain
- code
- data
- documentation in a standardised collection format
- can be installed by users of R, typically via a centralised software repository such as CRAN

https://roelverbelen.netlify.app/resources/r/packages/

Installing and Loading R Packages

1. Check available R package
2. Getting list of all installed packages
3. Install a new Package
   - Install directly from CRAN
   - Install package manually
4. Load package to library

```r
# Get the list of installed packages
installed.packages()

# Install package
install.packages()

# Import package
library()

# get all packages currently loaded in the R environment.
search()

# Check installed packages location
.libPaths()

# Update package
update.packages()
```

14

Classwork 1: Download packages

Search and download these packages:

- tidyverse

- readr

Help Files and Function Documentation

```
# Access the help file
?mean


# If unsure of the precise name
# search doc across all
installed packages
??mean
```

Saving Work and Exiting R

An R **workspace image** contains all the information held in the R session at the time of exit and is saved as a .RData file
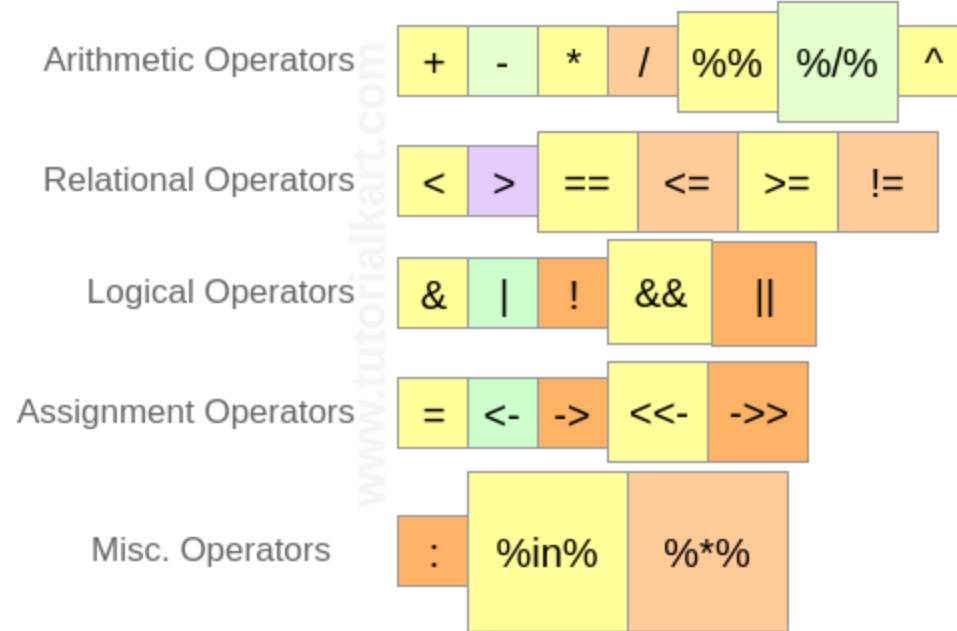
```
# Save current workspace
save.image(file="mysession.RData")

# exit R
q()

# Load workspace
load('myession.RData')
```

Basic R

Operators in R

Arithmetic Operators	+	-	*	/	%%	%/%	^

Relational Operators	<	>	==	<=	>=	!=

Logical Operators	&	|	!	&&	||

Assignment Operators	=	<-	->	<<-	->>

Misc. Operators	:	%in%	%*%

Operators in R

Arithmetic Operators

```
> 1 + 3
[1] 4
```

```
> 1/2
[1] 0.5
```

+ **−**

× **÷**

Logarithms and Exponentials

```
> log(10)
[1] 2.302585
> log10(10)
[1] 1
> exp(x=3)
[1] 20.08554
> sqrt(4)
[1] 2
```

Operators in R

Arithmetic Operators

$$10^2 + \frac{3 \times 60}{8} - 3$$

```
R> 10^2+3*60/8-3
[1] 119.5
```

$$\frac{5^3 \times (6-2)}{61-3+4}$$

```
R> 5^3*(6-2)/(61-3+4)
[1] 8.064516
```

$$2^{2+1} - 4 + 64^{-2^{2.25-\frac{1}{4}}}$$

```
R> 2^(2+1)-4+64^((-2)^(2.25-1/4))
[1] 16777220
```

$$\left(\frac{0.44 \times (1-0.44)}{34}\right)^{\frac{1}{2}}$$

```
R> (0.44*(1-0.44)/34)^(1/2)
[1] 0.08512966
```

Performing calculation in R

Operators in R

Arithmetic Operators - The Modulo and Integer Division

Modulo Operator (%%)
Returns the remainder of the division between two numbers.

```
# Basic Modulo Operation
> remainder <- 10 %% 3

# Even or Odd Check
is_even <- 4 %% 2
is_odd <- 5 %% 2
```

Integer Division Operator (%/%)
Returns the integer part of the division between two numbers.

```
# Basic Integer Division
quotient <- 10 %/% 3

# Integer Division of Even
Number
even_division <- 8 %/% 2

# Division Resulting in Zero
zero_quotient <- 2 %/% 3
```

Operators in R

a. Using R, verify that

$$\frac{6a + 42}{3^{4.2-3.62}} = 29.50556$$

when $a = 2.3$.

b. Which of the following squares negative 4 and adds 2 to the result?
   i. (-4)^2+2
   ii. -4^2+2
   iii. (-4)^(2+2)
   iv. -4^(2+2)

c. Using R, how would you calculate the square root of half of the average of the numbers 25.2, 15, 16.44, 15.3, and 18.6?

d. Find $\log_e 0.3$.

e. Compute the exponential transform of your answer to (d).

f. Identify R's representation of −0.00000000423546322 when printing this number to the console.

Operators in R

Comparison Operators

→ Compare values and return logical values (TRUE or FALSE).

```
# Equal to (==)
5 == 5   # Result: TRUE
5 == 3   # Result: FALSE


# Not equal to (!=)
5 != 3   # Result: TRUE
5 != 5   # Result: FALSE


# Greater than (>)
7 > 5   # Result: TRUE
5 > 7   # Result: FALSE
```

```
# Less than (<)
3 < 5   # Result: TRUE
5 < 3   # Result: FALSE


# Greater than or equal to (>=)
5 >= 5   # Result: TRUE
7 >= 5   # Result: TRUE
5 >= 7   # Result: FALSE


# Less than or equal to (<=)
5 <= 5   # Result: TRUE
3 <= 5   # Result: TRUE
5 <= 3   # Result: FALSE
```

- Use comparison operators to filter data frames or vectors based on conditions.
- Implement logic using if statements that depend on comparisons.

Operators in R

Logical Operators

Logical operators are used to perform element-wise logical operations and return logical values (TRUE or FALSE).
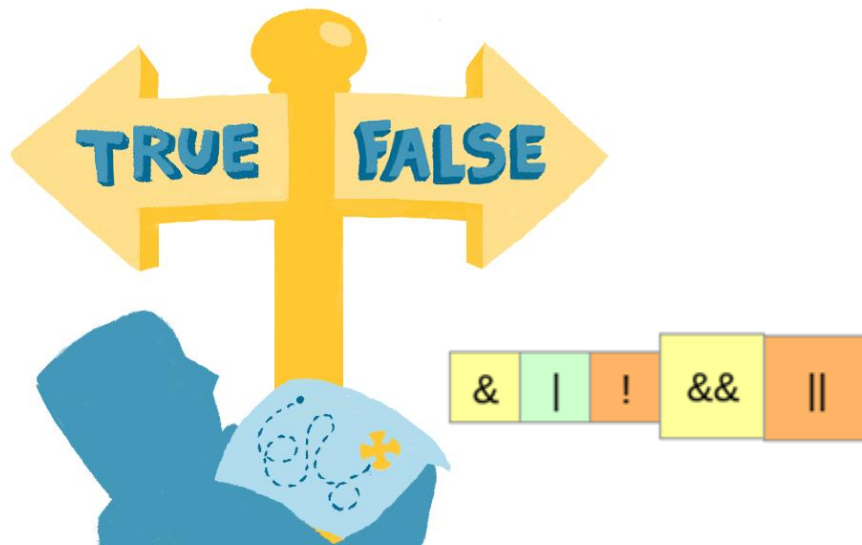
Operators in R

Logical Operators

AND (& and &&)

- Returns **TRUE** if both operands are **TRUE**.
- The single & is used for element-wise operations, while && is used for the first element of each vector only.

```
# AND (& and &&)
> 5>3 & 5==3
[1] FALSE


# Element-wise AND
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)
# Result: TRUE FALSE FALSE


# First element AND
c(TRUE, FALSE, TRUE) && c(TRUE, TRUE, FALSE)
(TRUE && FALSE)   # Result: FALSE
(TRUE && TRUE)    # Result: TRUE
```

Operators in R

Logical Operators

OR (| and ||)

- Returns TRUE if at least **one** operand is TRUE.
- The single | is used for element-wise operations, while || is used for the first element of each vector only.

```
# Example
> 5 == 3 | 5 == 5
[1] TRUE

# Element-wise OR
c(TRUE, FALSE, TRUE) | c(FALSE, TRUE, FALSE)
# Result: TRUE TRUE TRUE

# First element OR
(TRUE || FALSE)   # Result: TRUE
(FALSE || FALSE) # Result: FALSE
```

Operator in R

Logical Operators

NOT (!)

→ Returns the opposite logical value of the operand.

```
# Example
!TRUE  # Result: FALSE
!FALSE # Result: TRUE
```

Logical operators are essential for combining multiple conditions in R, enabling powerful data manipulation and control flow operations.

Operator in R

Classwork 3: Logical operators

https://colab.research.google.com/drive/1eyfgpWcLOe_8NNqac9B20I0eO-C0qdq-?usp=sharing

Filter this table
- Age > 30
- Income greater than 50000$

Assignment Operators

Assignment (<- or =)

Purpose: Assignment operators are used to assign values to variables in R.
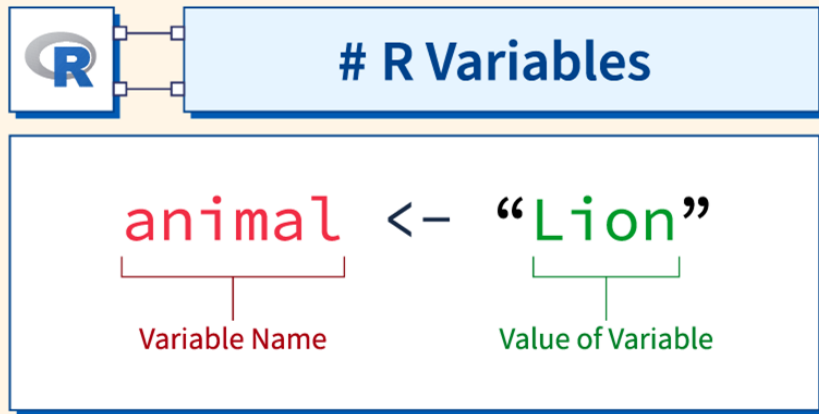

```
# Example
# Assign values to variables
name <- "Duy"
friend = "Minh"
"Loi" -> teacher
```

Assignment operators are fundamental in R, enabling you to store and manipulate data efficiently.

Assignment Operators

```
# Assign a value for x
> x <- -5
> x
[1] -5

# Assign another value for x
> x = x + 1 # this overwrites the previous value of x
> x
[1] -4

# Doing math with x and assign to y
> mynumber = 45.2
> y <- mynumber*x
> y
[1] -180.8
```

Assignment Operators

Classwork 4: Assignment Operator

https://colab.research.google.com/drive/1eyfgpWcLOe_8NNqac9B20I0eO-C0qdq-?usp=sharing

Make a sentence using these variables

```
# Example
# Assign values to variables
name <- "Duy"
friend = "Minh"
"Loi" -> teacher
```

Operators in R

Some special values (Inf, NaN, NA, NULL)

Inf (Infinity)

Represents infinity in R.

```
1 / 0
[1] Inf

-1 / 0
[1] -Inf
```

```
x <- c(10, Inf, 20)
sum(x)
[1] Inf
```

Operators in R

Some special values (Inf, NaN, NA, NULL)

NaN (Not a Number)

Represents undefined or unrepresentable numerical results.

```
0 / 0
[1] NaN


sqrt(-1)
[1] NaN
```

```
y <- c(1, NaN, 2)
is.nan(y)
[1] FALSE TRUE FALSE
```

- NaN is used to indicate a value that cannot be defined mathematically.

- Functions like `is.nan()` can be used to identify NaN values.

Operators in R

Some special values (Inf, NaN, NA, NULL)

NA (Not Available)

Represents missing values in R.

```
z <- c(1, 2, NA, 4)
mean(z)
[1] NA

mean(z, na.rm = TRUE)
[1] 2.333

is.na(z)
[1] FALSE FALSE TRUE FALSE
```

- NA can be of different types (numeric, character, etc.).

- Functions like is.na() help identify missing values.

Operators in R

Some special values (Inf, NaN, NA, NULL)

NULL (Empty or Undefined Value)

Represents the absence of a value or an undefined value.

```
my_list <- list(a = 1, b = NULL)
length(my_list)
[1] 2

print(NULL)
[1] NULL
```

- NULL is different from NA and NaN.

- Used to indicate that an object does not exist.

Operators in R

Some special values (Inf, NaN, NA, NULL)

Special Value	Meaning	Example Usage
Inf	Positive or negative infinity	1/0, -1/0
NaN	Undefined or unrepresentable value	0/0, sqrt(-1)
NA	Missing value	mean(c(1, NA), na.rm=TRUE)
NULL	No value or undefined value	list(a = 1, b = NULL)

Data Types and Structures

Data Types and Structures

Data structures

01	Vectors
02	Lists
03	Matrices
04	DataFrame
05	Factors

Vector

Matrix

rows

columns
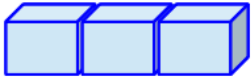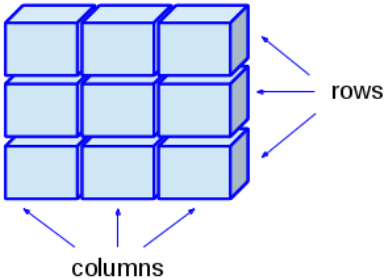
Array
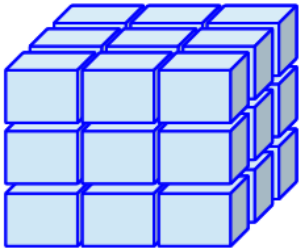
Data Frame
(Table)
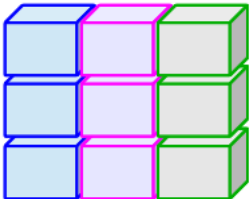
Lists
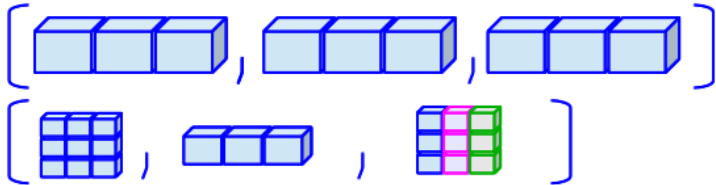
Data Types and Structures
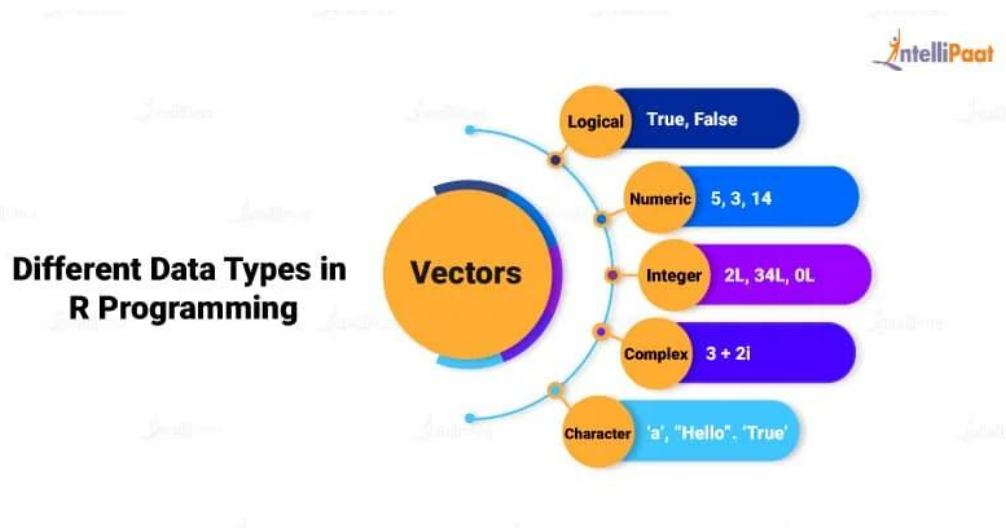
Vector


```r
# Numeric
numeric_vector <- c(1, 2,
3.5, 4.8, 6)

# Character
character_vector <-
c("apple", "banana",
"cherry")

# Logical
logical_vector <- c(TRUE,
FALSE, TRUE, FALSE)
```

Data Types and Structures

Vector-Oriented Behavior

```
# Creating a vector

vector1 <- c(329, 45, 12, 28)
```

values	329	45	12	28
indices	*1*	*2*	*3*	*4*

a[1]   a[2:3]   a[c(2,4)]
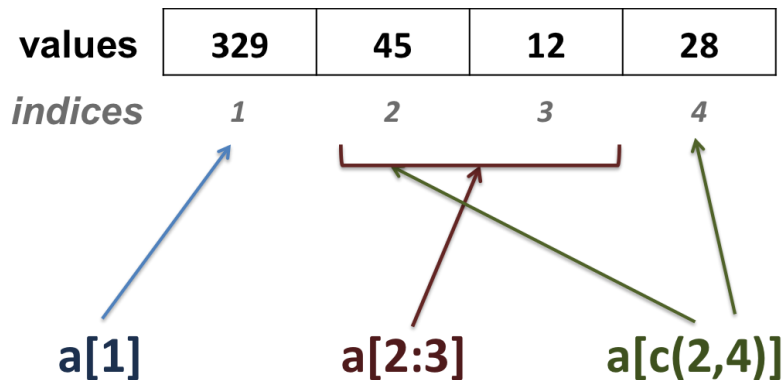
Vector carries out operations on multiple elements simultaneously with speed and efficiency.

This vectororiented, vectorized, or element-wise behavior is a **key feature of the language**

Vector Operation

Vector Arithmetic

Element-wise Operations:

```
# Numeric vector
numbers <-c(1,2,3,4,5)
# Character vector
names<-c("Alice","Bob","Charlie")

# Addition
result <- numbers + 2
print(result)
[1] 3 4 5 6 7

# Multiplication
result <- numbers *2 print(result)
[1] 2 4 6 8 10
```

Vector-Vector Operations

```
# Adding two vectors
vector1 <-c(1,2,3)
vector2 <-c(4,5,6)
result <- vector1 + vector2
print(result)

[1] 5 7 9
```

Vector Operation

Vector Indexing and Subsetting

Accessing Elements:

```
# Access the second element
second_element <- numbers[2]
print(second_element)
[1] 2
```

Subsetting:

```
# Get a subset of the first three
elements
subset_vector <- numbers[1:3]
print(subset_vector)
[1] 1 2 3
```

Logical Subsetting

```
# Get elements greater than 3
gt_than_3 <- numbers[numbers >3]
print(greater_than_three)
[1] 4 5
```

Vector Operation

Vector Naming

```r
# Name the elements of the vector
names(numbers)<-c("First","Second","Third","Fourth","Fifth")
print(numbers)
[1] First Second Third Fourth Fifth # 1 2 3 4 5
```

Vector Type

```r
# Get the type of the vector
vector_type <- typeof(numbers)
print(vector_type)
[1] "double"
```

Combining Vectors

```r
# Combine vectors
combined_vector <-c(vector1, vector2)
print(combined_vector)
[1] 1 2 3 4 5 6
```

Vector Operation

The Recycling Rule

→ How R handles operations between vectors of unequal lengths.

→ R will "recycle" the shorter vector by repeating its elements until it matches the length of the longer vector.

```r
# Shorter vector is recycled to match the length of the longer vector
short_vector <-c(1,2)
long_vector <-c(10,20,30,40)
result <- long_vector + short_vector
print(result)
[1] 11 22 31 42
```

Vector Operation
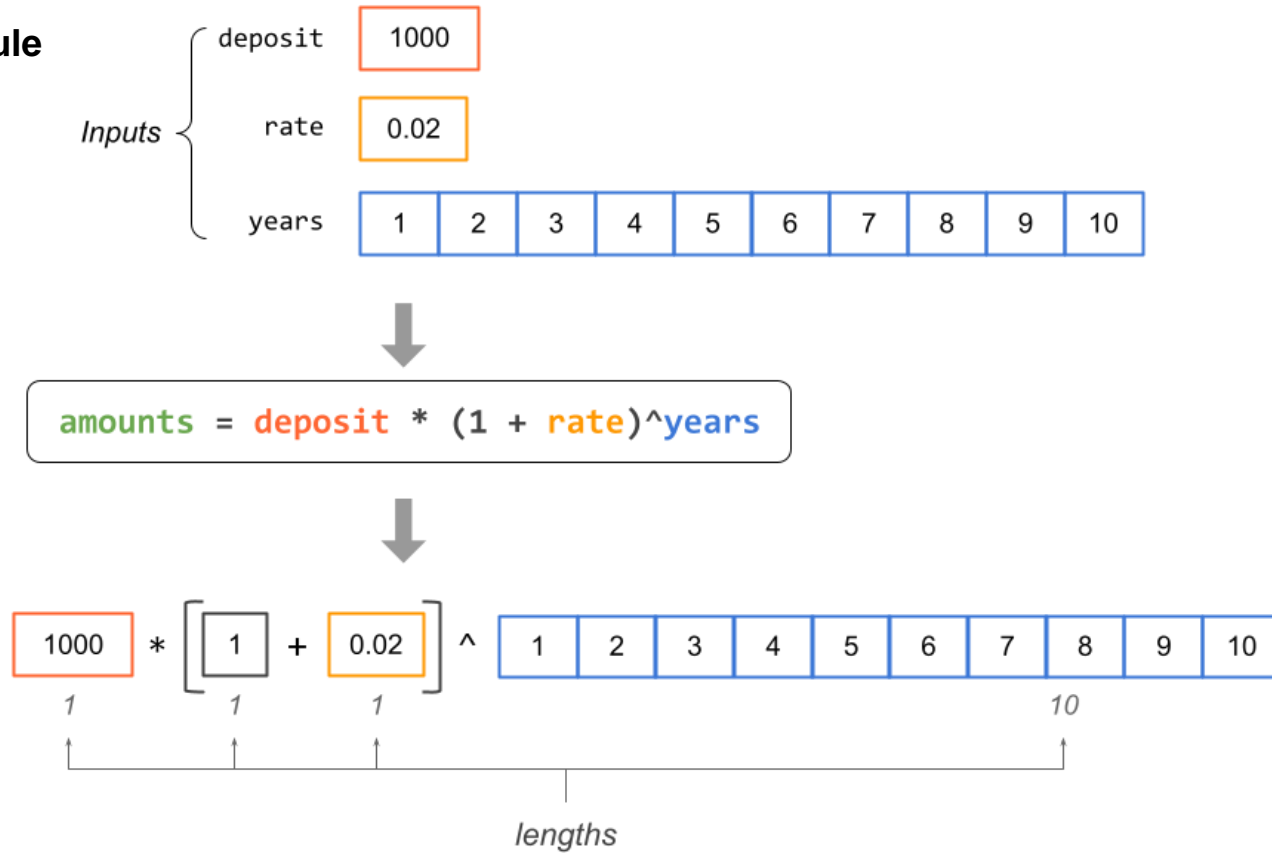
The Recycling Rule

Vector Operation

$$\texttt{amounts = deposit * (1 + rate)\^{}years}$$

The Recycling Rule

This is what R will do "behind the scenes"

https://www.gastonsanchez.com/R-coding-basics/vectors4.html#recycling
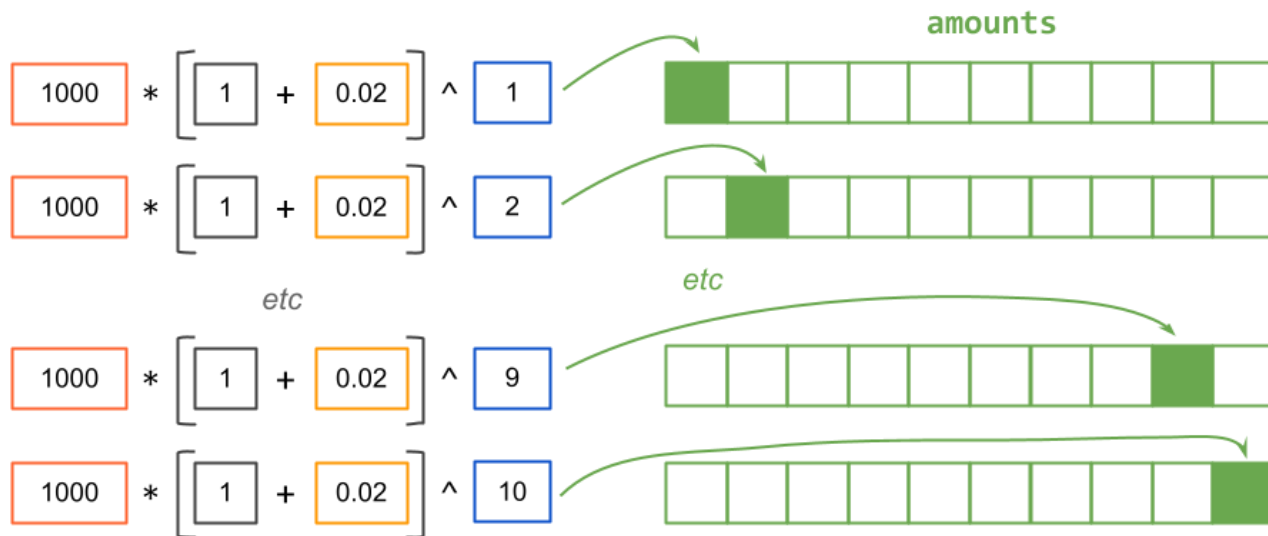
Vector Operation

```
# Sequences with seq()
> seq(from=3, to=27, by=3)
[1]   3   6   9  12  15  18  21  24  27

# Repetition with rep()
> rep(x=1,times=4)
[1] 1 1 1 1
> rep(x=c(3,62,8.3),times=3)
[1]   3.0 62.0   8.3   3.0 62.0   8.3   3.0 62.0 8.3

# Sorting with sort()
> sort(x=c(2.5,-1,-10,3.44),decreasing=FALSE)
[1] -10.00  -1.00   2.50    3.44
> sort(x=c(2.5,-1,-10,3.44),decreasing=TRUE)
[1]   3.44   2.50  -1.00 -10.00

# Finding a Vector length with length()
> length(x=c(3,2,8,1))
[1] 4
```

Special functions: Sequences, Repetition, Sorting, and Lengths

Vector Operation

Subsetting and Element Extraction

```
> myvec <- c(5,-2.3,4,4,4,6,8,10,40221,-8)
> length(x=myvec)
[1] 10

> myvec[1]
[1] 5

> foo <- myvec[2]
> foo
[1] -2.3

> myvec[length(x=myvec)]
[1] -8
```

Vectors in R

Index →	1	2	3	4	5	6	7	8	9	10
Values →	10	20	30	40	50	60	70	80	90	100

GG

https://www.geeksforgeeks.org/r-vector/

Vector Operation

Factors

- Represent categorical data in R
- Store both the values of the data and the corresponding levels
- Unique values

```
# Creating a factor from a character vector
colors <-c("red","green","blue","red","green")
color_factor <- factor(colors) print(color_factor)

[1] red green blue red green
Levels: blue green red
```

```
# Specifying the order of levels
ordered_factor <- factor(colors, levels =c("red","green","blue"))
print(ordered_factor)

[1] red green blue red green
Levels: red green blue
```

Vector Operation

Classwork 5: Replicate all the Vector operation codes above

Matrix and array
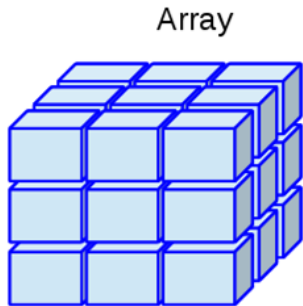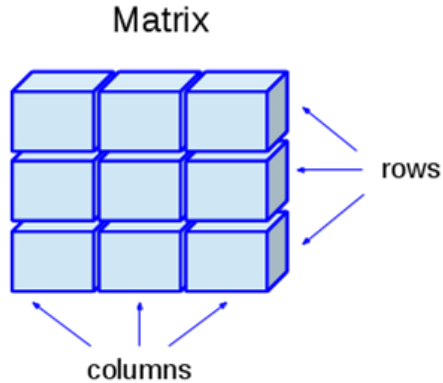
Matrix

Matrix

Definition: A matrix is a two-dimensional (2D) data structure in R where all elements are of the **same** data type (numeric, character, or logical).

Structure: Consists of rows and columns.

Array

Array

Definition: An array is a multi-dimensional data structure in R that can have more than two dimensions. All elements must be of the **same** type.

Structure: Arrays can be thought of as matrices extended to more dimensions.

Matrix and array

Matrix

rows

columns
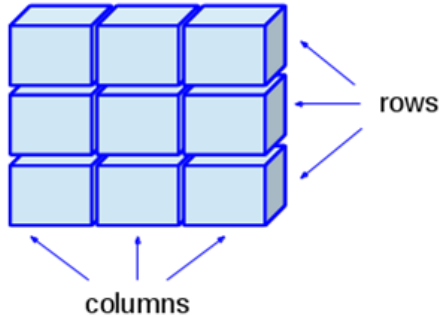
Array

Matrix

```
# Create a 3x3 numeric matrix
mat <- matrix(1:9, nrow =3, ncol =3)
print(mat)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```
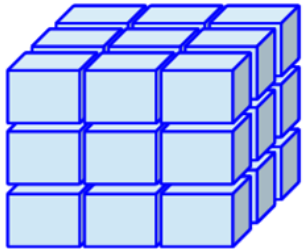
Array

```
# Create a 3x3x2 array
arr <- array(1:18,dim=c(3,3,2))
print(arr)
```

```
, , 1

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

```
, , 2

     [,1] [,2] [,3]
[1,]  10  13  16
[2,]  11  14  17
[3,]  12  15  18
```

53

Matrix

Matrix operation

```
# Create matrix filled by row
mat_by_row <- matrix(1:9, nrow
=3, byrow =TRUE)
print(mat_by_row)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Element-wise Operations

```
# Multiply matrix by 2
mat_times_two <- mat *2
print(mat_times_two)
     [,1] [,2] [,3]
[1,]    2    8   14
[2,]    4   10   16
[3,]    6   12   18
```

Matrix Multiplication

```
# Matrix multiplication
mat_mult <- mat %*% t(mat)
print(mat_mult)

      [,1] [,2] [,3]
[1,]    66   78   90
[2,]    78   93  108
[3,]    90  108  126
```

Matrix

Subsetting Matrix

Extracting Elements:

```
# Extract element from 2nd row, 3rd column
element <- mat[2,3]
print(element)
 [1] 8
```

Extracting Rows/Columns:

```
# Extract entire 1st row
row <- mat[1,]
print(row)
 [1] 1 4 7
```

Matrix

Common Matrix Functions

Transpose

```
# Transpose a matrix
mat <- matrix(1:9, nrow=3, ncol=3)
print(mat)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

t_mat <- t(mat)
print(t_mat)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

Matrix Dimensions

```
# Get matrix dimensions
dim_mat <-dim(mat)
print(dim_mat)

[1] 3 3
```

Matrix

Classwork 6: Create a microbiome feature matrix

Background:
In microbiome research, data is often organized in **matrix** where rows represent different samples (e.g., from different patients or environmental sites) and columns represent various attributes like bacterial species, metadata (e.g., sample location, date), or calculated metrics (e.g., diversity indices).
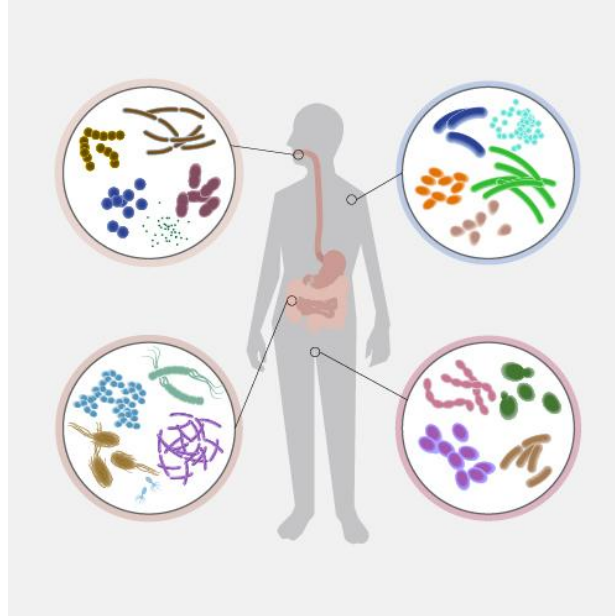
Matrix

Classwork 6: Create a microbiome feature matrix

1.1 Create a matrix named `microbiome_data` representing the abundance of 5 bacterial species across 4 different samples. Use the following data

```
microbiome_data <- matrix(c(23, 5, 0, 12, 9, 8, 15, 13, 7, 2,
14, 9, 6, 11, 1, 3, 8, 2, 10, 5), nrow = 4, byrow = TRUE)
```

1.2 Assign row names as `"Sample_1"`, `"Sample_2"`, `"Sample_3"`, and `"Sample_4"`, and column names as `"Species_1"`, `"Species_2"`, `"Species_3"`, `"Species_4"`, and `"Species_5"`.

	Species_1	Species_2	Species_3	Species_4	Species_5
Sample_1	23	5	0	12	9
Sample_2	8	15	13	7	2
Sample_3	14	9	6	11	1
Sample_4	3	8	2	10	5

Matrix

Classwork 6: Create a microbiome feature matrix

Part 2: Basic Matrix Operations

2.1 Extract the abundance data for `"Species_3"` across all samples.

2.2 Extract the data for `"Sample_2"` across all species.

2.3 Calculate the total abundance for each sample. (Use rowSums function)

2.4 Calculate the average abundance for each species across all samples. (Use colMeans function)

Matrix

Classwork 6: Create a microbiome feature matrix

Part 3: Advanced Matrix Operations

3.1 Transpose the `microbiome_data` matrix to switch rows and columns.

3.2 Identify the sample with the highest abundance of `"Species_1"`.

- **Hint**: Use the `which.max()` function to find the index.

3.3 Add a new species (`"Species_6"`) with the following abundance data: `[7, 10, 3, 5]`.

- **Hint**: Use the `cbind()` function to add a new column.

Data Frame

Data Frame
(Table)
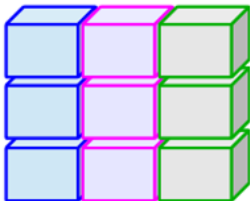
1	"S"	TRUE
7	"A"	FALSE
3	"U"	TRUE
numeric	character	logical

Definition:
A data frame is a table or a 2-dimensional array-like structure in R, where each column can contain **different types of data** (numeric, character, factor, etc.).

Structure:
Similar to a spreadsheet or SQL table, with rows representing observations and columns representing variables.

Data Frame

Creating a Data Frame

```r
# Create a data frame with three columns
df <- data.frame(ID =1:4,
                 Name=c("Alice","Bob","Charlie","Diana"),
                 Score =c(85,92,88,76))
print(df)
```

```
  ID    Name Score
1  1   Alice    85
2  2     Bob    92
3  3 Charlie    88
4  4   Diana    76
```

Data Frame

Accessing Data in a Data Frame

Using $ to Access Columns:

```
# Access the 'Name'
columnnames<- df$Name print(names)

[1] "Alice" "Bob" "Charlie" "Diana"
```

Using Indexing

```
# Access the element in the 2nd row, 3rd column
element <- df[2,3] print(element)

[1] 92
```

Data Frame

Common Data Frame Operations

Adding a New Column

```
# Add a new column 'Passed'

df$Passed <- df$Score >80
print(df)
```

```
  ID    Name Score Passed
1  1   Alice    85   TRUE
2  2     Bob    92   TRUE
3  3 Charlie    88   TRUE
4  4   Diana    76  FALSE
```

Subsetting Data Frames:

```
# Subsetting a dataframe with
condition

high_scores <- df[df$Score >80,]
print(high_scores)
```

```
  ID    Name Score Passed
1  1   Alice    85   TRUE
2  2     Bob    92   TRUE
3  3 Charlie    88   TRUE
```

Data Frame

Combining Data Frames

Row Binding

```
# Combine data frames by adding rows

df_new <- data.frame(ID=5,
                     Name="Eve",
                     Score=90)
combined_df <- rbind(df, df_new)
print(combined_df)
```

	ID	Name	Score	Passed
1	1	Alice	85	TRUE
2	2	Bob	92	TRUE
3	3	Charlie	88	TRUE
4	4	Diana	76	FALSE
5	5	Eve	90	TRUE

Column Binding

```
# Combine data frames by adding
columns

extra_info <-
data.frame(Age=c(23,25,22,21,24))
full_df <- cbind(combined_df,
                 extra_info)
print(full_df)
```

	ID	Name	Score	Passed	Age
1	1	Alice	85	TRUE	23
2	2	Bob	92	TRUE	25
3	3	Charlie	88	TRUE	22
4	4	Diana	76	FALSE	21
5	5	Eve	90	TRUE	24

Data Frame

Viewing and Inspecting Data Frames

```
# Viewing data
View(df)


# Explore the structure of the data
str(df)
'data.frame':    4 obs. of  4 variables:
 $ ID    : int  1 2 3 4
 $ Name  : chr  "Alice" "Bob" "Charlie" "Diana"
 $ Score : num  85 92 88 76
 $ Passed: logi  TRUE TRUE TRUE FALSE
```

Data Frame

Summary Statistics

To get a summary of each column.

```
summary(df)

      ID              Name                  Score              Passed
 Min.   :1.00    Length:4            Min.   :76.00     Mode :logical
 1st Qu.:1.75    Class :character    1st Qu.:82.75     FALSE:1
 Median :2.50    Mode  :character    Median :86.50     TRUE :3
 Mean   :2.50                        Mean   :85.25
 3rd Qu.:3.25                        3rd Qu.:89.00
 Max.   :4.00                        Max.   :92.00
```

Data Frame

Subsetting and Filtering Data

Subset Rows Based on Conditions

```
# Get rows where Score is
greater than 80

high_scores <- df[df$Score
>80,]
print(high_scores)
```

	ID	Name	Score	Passed
1	1	Alice	85	TRUE
2	2	Bob	92	TRUE
3	3	Charlie	88	TRUE

Select Specific Columns

```
# Select only the 'Name' and 'Score'
columns

name_score <- df[,c("Name","Score")]
print(name_score)
```

	Name	Score
1	Alice	85
2	Bob	92
3	Charlie	88
4	Diana	76

Data Frame

Adding and Modifying Columns

Add a New Column

```
# Add a column indicating if the score
is above average
df$Above_Average <-
df$Score > mean(df$Score)
print(df)
```

	ID	Name	Score	Passed	Above_Average
1	1	Alice	85	TRUE	FALSE
2	2	Bob	92	TRUE	TRUE
3	3	Charlie	88	TRUE	TRUE
4	4	Diana	76	FALSE	FALSE

Modify an Existing Column

```
# Adjust the score by adding 5 points
to each student

df$Score <- df$Score + 5
print(df)
```

	ID	Name	Score	Passed	Above_Average
1	1	Alice	90	TRUE	FALSE
2	2	Bob	97	TRUE	TRUE
3	3	Charlie	93	TRUE	TRUE
4	4	Diana	81	FALSE	FALSE

Data Frame

Sorting Data Frames

Sort by a Single Column

```
# Sort the data frame by 'Score' in
descending order
df_sorted <- df[order(-df$Score),]
print(df_sorted)
```

	ID	Name	Score	Passed	Above_Average
2	2	Bob	97	TRUE	TRUE
3	3	Charlie	93	TRUE	TRUE
1	1	Alice	90	TRUE	FALSE
4	4	Diana	81	FALSE	FALSE

Sort by Multiple Columns

```
# Sort by 'Passed' (descending) and
then by 'Score' (ascending)

df_sorted_multi <- df[order(-df$Passed,
df$Score),]
print(df_sorted_multi)
```

	ID	Name	Score	Passed	Above_Average
1	1	Alice	90	TRUE	FALSE
3	3	Charlie	93	TRUE	TRUE
2	2	Bob	97	TRUE	TRUE
4	4	Diana	81	FALSE	FALSE

Data Frame

Combining Data Frames

Row Binding

```
# Bind new dataframe rows to an existed one
new_students <- data.frame(ID = 5,
                           Name = "Eve",
                           Score = 89,
                           Passed = TRUE,
                           Above_Average = FALSE)
df_combined <- rbind(df, new_students)
print(df_combined)
```

```
  ID    Name Score Passed Above_Average
1  1   Alice    90   TRUE         FALSE
2  2     Bob    97   TRUE          TRUE
3  3 Charlie    93   TRUE          TRUE
4  4   Diana    81  FALSE         FALSE
5  5     Eve    89   TRUE         FALSE
```


https://www.gastonsanchez.com/intro2cwd/arrays.html

71

Data Frame

Combining Data Frames

Column Binding

```
# Add a new column for student
Age ages <- data.frame(Age =
c(23,25,22,21,24))
df_with_age <- cbind(df_combined, ages)
print(df_with_age)
```

	ID	Name	Score	Passed	Above_Average	Age
1	1	Alice	90	TRUE	FALSE	23
2	2	Bob	97	TRUE	TRUE	25
3	3	Charlie	93	TRUE	TRUE	22
4	4	Diana	81	FALSE	FALSE	21
5	5	Eve	89	TRUE	FALSE	24

column binding

`mat3 <- cbind(mat1, mat2)`

https://www.gastonsanchez.com/intro2cwd/arrays.html

Data Frame

Removing or Renaming Columns

Remove a Column

```
# Remove the 'Passed' column
df_no_passed <-
df[,!(names(df)%in%"Passed")]
print(df_no_passed)
```

	ID	Name	Score	Above_Average
1	1	Alice	90	FALSE
2	2	Bob	97	TRUE
3	3	Charlie	93	TRUE
4	4	Diana	81	FALSE

Rename a Column

```
# Rename 'Score' to 'Final_Score'

names(df)[names(df)=="Score"]<-"Final_Score"
print(df)
```

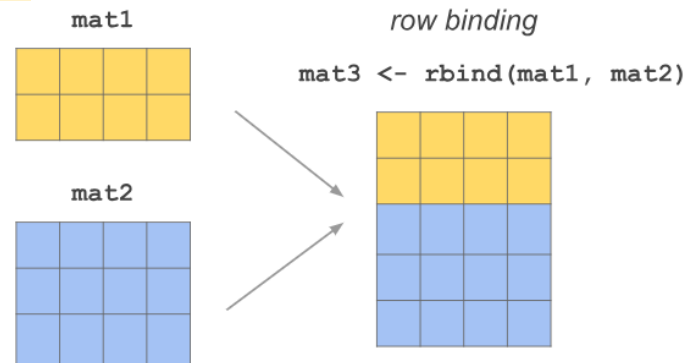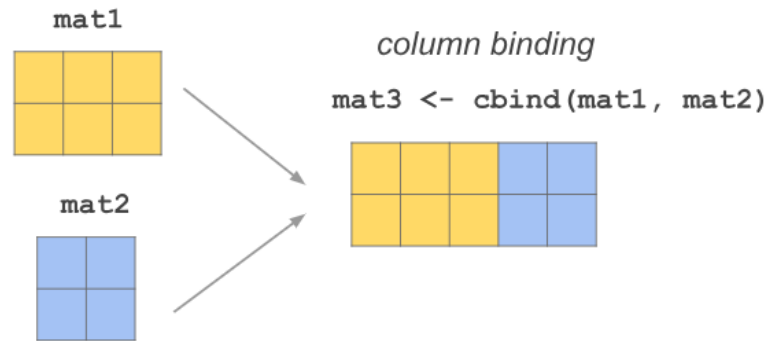|	ID	Name	Final_Score	Passed	Above_Average
1	1	Alice	90	TRUE	FALSE
2	2	Bob	97	TRUE	TRUE
3	3	Charlie	93	TRUE	TRUE
4	4	Diana	81	FALSE	FALSE

Data Frame

Merging Data Frames

Key Variable	Variable A	Variable B	Variable C	Variable D
1	3.1	7.3	1	23
2	4.5	9.9	0	21
3	5.0	8.5	0	44
4	1.0	8.4	1	50

Key Variable	Variable E	Variable F	Variable G	Variable H
1	86	Red	4.9	19
2	95	Green	5.0	20
3	78	Red	5.0	14
4	91	Blue	4.1	13

Key Variable	Variable A	Variable B	Variable C	Variable D	Variable E	Variable F	Variable G	Variable H
1	3.1	7.3	1	23	86	Red	4.9	19
2	5.0	8.5	0	44	95	Green	5.0	20
3	5.0	8.5	0	44	78	Red	5.0	14
4	1.0	8.4	1	50	91	Blue	4.1	13

https://rforhr.com/join.html

Data Frame

Merging Data Frames

```r
# Merge two data frames by the 'ID' column

df_info <- data.frame(ID =1:4,
                      Gender =
                      c("F","M","M","F"))
df_merged <- merge(df, df_info, by ="ID")
print(df_merged)
```

	ID	Name	Score	Passed	Above_Average	Gender
1	1	Alice	90	TRUE	FALSE	F
2	2	Bob	97	TRUE	TRUE	M
3	3	Charlie	93	TRUE	TRUE	M
4	4	Diana	81	FALSE	FALSE	F

Data Frame

Other Key Functions

```
nrow(df):        Number of rows.

ncol(df):        Number of columns.

dim(df):         Dimensions (rows, columns).

names(df):       Column names.
```

Data Frame

Classwork 7: Replicate all the dataframe operation codes above

Data Frame

Classwork 8: Working with Microbiome Data Frames in R

Exercise Objectives:

1. Create and manipulate a data frame representing microbiome data.
2. Perform basic operations such as subsetting, filtering, and summarizing data.

Data Frame

Classwork 8: Working with Microbiome Data Frames in R

Part 1: Creating a Data Frame

1.1 Create a data frame named microbiome_df containing the following data

```
microbiome_df <- data.frame(
                          Sample_ID = c("Sample_1", "Sample_2", "Sample_3","Sample_4"),
                          Location = c("Gut", "Skin", "Mouth", "Gut"),
                          Species_1 = c(23, 8, 14, 3),
                          Species_2 = c(5, 15, 9, 8),
                          Species_3 = c(0, 13, 6, 2),
                          Species_4 = c(12, 7, 11, 10),
                          Species_5 = c(9, 2, 1, 5) )
```

Sample_ID	Location	Species_1	Species_2	Species_3	Species_4	Species_5
Sample_1	Gut	23	5	0	12	9
Sample_2	Skin	8	15	13	7	2
Sample_3	Mouth	14	9	6	11	1
Sample_4	Gut	3	8	2	10	5

1.2 Display the structure of the data frame to understand its composition.

Data Frame

Classwork 8: Working with Microbiome Data Frames in R

Part 2: Basic Data Frame Operations

2.1 Extract the data for `"Sample_3"` (all columns).

2.2 Extract the data for `Species_2` across all samples.

2.3 Calculate the total bacterial abundance for each sample (sum of `Species_1` to `Species_5`).

Data Frame

Classwork 8: Working with Microbiome Data Frames in R

Part 3: Filtering and Subsetting

3.1 Filter the data frame to include only samples from the `"Gut"` location.

3.2 Subset the data frame to include only `"Sample_ID"`, `"Location"`, and `"total_abundance"` columns.

Part 4: Summarizing Data

4.1 Calculate the mean abundance for `"Species_1"` across all samples.

4.2 Find the sample with the highest total abundance.

4.3 Count the number of samples from each location.

Data Frame

Classwork 8: Working with Microbiome Data Frames in R

Part 5: Adding New Data

5.1 Add a new column to the data frame representing a calculated metric, such as the ratio of `Species_1` to `total_abundance`.

5.2 Add a new sample to the data frame with the following information:
- **Sample_ID**: `"Sample_5"`
- **Location**: `"Skin"`
- **Species_1 to Species_5**: `[6, 12, 4, 8, 3]`

List
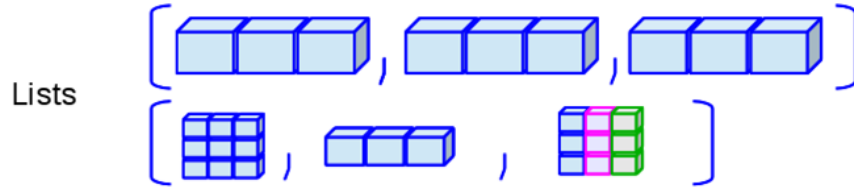
Lists

Definition:
A list is a versatile data structure in R that can contain elements of different types (vectors, data frames, functions, etc.).

Structure:
Lists can hold any number of elements, each of which can be of a different type and length.

List

Creating a List

```r
# Create a list with different types of elements
my_list <- list(Name = "Alice",
                Age = 25,
                Scores = c(85, 92, 88),
                Passed = TRUE,
                Info = data.frame(Subject =
                c("Math", "Science"), Grade =
                c("A", "B")) )
print(my_list)
```

```
$Name
[1] "Alice"


$Age
[1] 25


$Scores
[1] 85 92 88


$Passed
[1] TRUE


$Info
  Subject Grade
1    Math     A
2 Science     B
```

List

Accessing Elements in a List

Using **$** or **[[ ]]**:

```
# Access the 'Name' element
name <- my_list$Name
print(name)
[1] "Alice"


# Access the 'Scores' element
scores <- my_list[["Scores"]]
print(scores)
[1] 85 92 88
```

```
# Access the first score
first_score <- my_list[["Scores"]][1]
print(first_score)
[1] 85


# Access the 'Grade' column in the
'Info' data frame
grades <- my_list$Info$Grade
print(grades)
[1] "A" "B"
```

List

Modifying Lists

Adding Elements

```
# Add a new element to the list
my_list$City <- "New York"
print(my_list)
```

Removing Elements

```
# Remove the 'Passed' element
my_list$Passed <- NULL
print(my_list)
```

Updating Existing Elements

```
# Update the 'Age' element
my_list$Age <- 26
print(my_list$Age)
[1] 26
```

List

Combining and Splitting Lists

Combining Lists

```r
# Combine two lists
another_list <- list(Hobbies =
c("Reading", "Traveling"))

combined_list <- c(my_list,
another_list)
print(combined_list)
```

```
$Name
[1] "Alice"


$Age
[1] 25


$Scores
[1] 85 92 88


$Passed
[1] TRUE


$Info
   Subject Grade
1    Math     A
2 Science     B


$Hobbies
[1] "Reading"   "Traveling"
```

List

Unlisting a List

```r
# Flatten the list into a vector
unlisted <- unlist(my_list)
print(unlisted)
```

Name	Age	Scores1	Scores2	Scores3
"Alice"	"25"	"85"	"92"	"88"
Passed	Info.Subject1	Info.Subject2	Info.Grade1	Info.Grade2
"TRUE"	"Math"	"Science"	"A"	"B"

List

List Operations

Length of a List

```
# Get the number of elements in the list
list_length <- length(my_list)
print(list_length) # Output: 5
```

Looping Through Lists

```
# Loop through list elements
for (element in my_list) {
    print(element)
}
```

List

When to Use Lists

- **Heterogeneous Data**: When you need to store elements of different types or structures.
- **Complex Objects**: When storing outputs from models, results from multiple analyses, or data that doesn't fit neatly into a vector or data frame.

Data structure summary

91
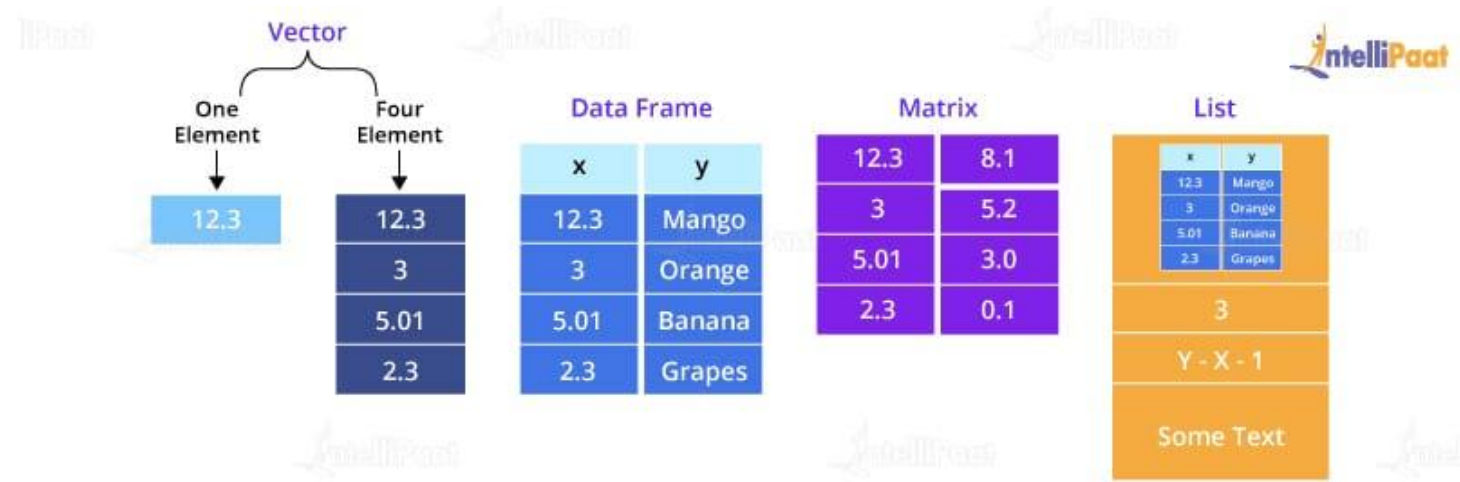
Save data

Why Save Data?

- **Preservation**: Store your processed data for future use.
- **Sharing**: Easily share data with others or transfer between projects.
- **Efficiency**: Avoid re-running expensive data processing steps.

Save data

Common Functions to Save Data

save()

Save one or more R objects (e.g., data frames, lists) to an R data file (`.RData` or `.rda`).

```
save(object1, object2, file = "mydata.RData")
save(df, list_of_results, file = "results.RData")
```

saveRDS()

Save a single R object to an RDS file (`.rds`).

```
saveRDS(df, file = "dataframe.rds")
```

Save data

Common Functions to Save Data

write.csv()

Save a data frame as a CSV file.

```
write.csv(df, file = "results.csv", row.names = FALSE)
```

write.table()

Save data to a text file with more control over formatting.

```
write.table(df, file = "results.txt", sep = "\t",
row.names = FALSE)
```

Save data

Loading Saved Data

load(): Load an R data file (`.RData` or `.rda`)

```
load("mydata.RData")
```

readRDS(): Load an RDS file.

```
df <- readRDS("dataframe.rds")
```

R functions

R Functions
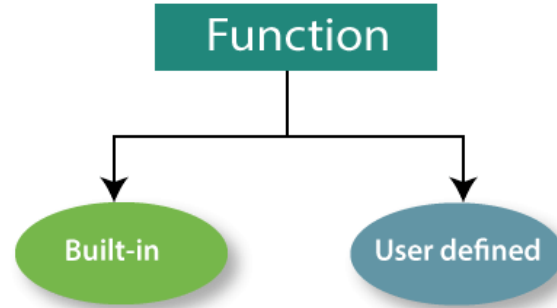
- Useful Built-in function
- Create an R function

https://www.statmethods.net/mana
gement/functions.html

https://iqss.github.io/dss-
workshops/R/Rintro/base-r-cheat-

R Functions

Useful Built-in function

Data Manipulation

- **subset()**: Extract subsets of data.
- **merge()**: Combine data frames by common columns or row names.
- **apply()**: Apply a function over the margins of an array or matrix.
- **tapply()**: Apply a function over subsets of a vector.
- **reshape()**: Reshape data between wide and long formats.
- **cut()**: Divide continuous variables into intervals.
- **aggregate()**: Compute summary statistics over subsets of data.

Statistical Analysis

- **summary()**: Provide a summary of an object.
- **cor()**: Calculate correlation between variables.
- **lm()**: Fit linear models.
- **table()**: Create a contingency table of counts.

R Functions

Useful Built-in function

Data Cleaning

- `na.omit()`: Remove missing values from an object.
- `is.na()`: Identify missing values.
- `duplicated()`: Identify duplicate elements.

Data Visualization

- `plot()`: Generic X-Y plotting.
- `hist()`: Create a histogram.
- `boxplot()`: Create a boxplot.
- `pairs()`: Create a matrix of scatterplots.

R Functions

Useful Built-in function

Utility Functions

- `str()`: Display the structure of an R object.
- `paste()`: Concatenate strings.
- `seq()`: Generate a sequence of numbers.
- `rep()`: Repeat elements of a vector.

R Functions

Apply function and its relatives

Purpose: To demonstrate the use of apply functions for efficient data manipulation in microbiome research.
Key Functions: `apply()`, `lapply()`, `sapply()`, `tapply()`, `mapply()`

Advantages of Using Apply Functions

- **Efficiency**: Faster execution by avoiding explicit loops.
- **Readability**: Clean and concise code.
- **Vectorization**: Leverages R's strength in vectorized operations.

R Functions

Apply function and its relatives

apply()

```
apply(X, MARGIN, FUN, ...)
```

```
# Compute row sums of a matrix
matrix_data <- matrix(1:9, nrow = 3)
apply(matrix_data, 1, sum)   # Sums of rows
```

Parameters:

- **X**: Array or matrix.
- **MARGIN**: An integer vector indicating which margins should be "retained". 1 indicates rows, 2 for columns.
- **FUN**: The function to be applied.

R Functions

Apply function and its relatives

lapply()

Usage: Apply a function over elements in a list, returning a list.

```
lapply(X, FUN, ...)
```

```
# Compute length of each string in a list

string_list <- list("apple", "banana",

"cherry")

lapply(string_list, nchar)
```

Parameters:

- **X**: List or vector.
- **FUN**: The function to apply.

R Functions

Apply function and its relatives

sapply()

Usage: Simplified version of `lapply()` that tries to simplify the result to a vector or matrix.

```
sapply(X, FUN, ..., simplify = TRUE)
```

```
# Compute square of each number

numbers <- 1:5

sapply(numbers, function(x) x^2)
```

Parameters:

- **X**: List or vector.
- **FUN**: The function to apply.

R Functions

Apply function and its relatives

tapply()

Usage: Apply a function over subsets of a vector, grouped by some other vector, usually a factor.

```
tapply(X, INDEX, FUN, ..., simplify = TRUE)
```

```
# Calculate mean weight by group

weights <- c(50, 60, 65, 70)

group <- factor(c("Male", "Female",

"Female", "Male"))

tapply(weights, group, mean)
```

Parameters:

- **X**: A vector.
- **INDEX**: A factor or a list of factors (the grouping variable).
- **FUN**: The function to apply.

R Functions

Create an R function

What is a Function?

- **Definition**: A function in R is a block of code designed to perform a specific task. It can take inputs, process them, and return outputs.
- **Purpose**: Functions allow for code reuse, organization, and abstraction of complex operations.

R Functions

Create an R function

Basic Structure

```r
function_name <- function(arg1, arg2, ...) {

    # Code block

    result <- arg1 + arg2 # Example operation

    return(result) # Return the result

}
```

Explanation:

- **function_name**: Name of your function.
- **function**: Keyword to define a function.
- **arg1, arg2, ...**: Arguments or parameters the function accepts.
- **{}**: Curly braces enclose the body of the function.
- **return(result)**: Specifies what the function should output

R Functions

Example: Creating a Simple Function

Sum of Two Numbers

```
sum_two_numbers <- function(a, b) {

    result <- a + b

    return(result)

}
# Using the function

sum_two_numbers(5, 3)

[1] 8
```

R Functions

Default Arguments

If no argument is provided, the function uses the default value.

```r
# Setting Default Values
greet <- function(name = "World") {

    message <- paste("Hello,", name)

    return(message)

}


# Calling the function
greet()

[1] "Hello, World"

greet("Duy")

[1] "Hello, Duy"
```

R Functions

Returning Multiple Values

A list is used to return multiple values from a function.

```r
# Using a List
stats <- function(x) {
    mean_val <- mean(x)
    sd_val <- sd(x)
    return(list(mean = mean_val, sd = sd_val))
}

# Calling the function
result <- stats(c(1, 2, 3, 4, 5))
print(result$mean)
[1] 3
print(result$sd)
[1] 1.581139
```

R Functions

Scope of Variables

Local vs Global Variables

```r
my_function <- function(x) {
    y <- x + 1 # 'y' is a local variable
    return(y)
}

y <- 10 # 'y' is a global variable
result <- my_function(5)
print(y)

[1] Output: 10
#(global 'y' is unchanged)
```

Variables defined inside a function are local to that function and do not affect global variables.

R Functions

Benefits of Using Functions

- **Modularity**: Break down complex tasks into smaller, manageable pieces.
- **Reusability**: Write code once and reuse it multiple times.
- **Maintainability**: Easier to debug and update code.
- **Abstraction**: Hide complex logic behind simple function calls.

Best Practices

- **Clear Naming**: Use descriptive names for functions and arguments.
- **Documentation**: Comment your functions and use # to explain what each part does.
- **Error Handling**: Validate inputs and handle potential errors inside your functions.

END