```r
# Assignment Operators
#  --------------------
# Used to store values inside variables

# <-   (Most common, rightward assignment operator)
hight <- c(1.75,1.76,1.82,1.67)


# ->   (Same as above, butleftward assignment operator)
 c(68, 78, 85, 75) -> weight


# =     (Also assigns, used in function arguments)
smoking_status = c("Yes", "No", "No", "Yes")







# -------------------
# Arithmetic Operators
# -------------------
# perform basic math i.e:
#   +  addition
#   -  subtraction
#   *  multiplication
#   /  division
#   ^  exponent (to the power of)

# let's BMI using weight and height

BMI  <-  weight/(height^2)
BMI

# vectorization
# R perform operation to every value in the vector









#  ------------------------
#  comparison Operators
#  ------------------------
# comparison operators ask logical questions about values .
# They return output as True or False
# They don't calculate answer
They compare values

#   >   greater than
BMI > 25
```

```
#   <   less than
BMI < 18.5
#   >=  greater than or equal to
#   <=  less than or equal to
#   ==  equal to
#   !=  not equal to

# In R
# yes = TRUE
# no  = FALSE









# --------------------
# Logical Operators
# --------------------
# Logical operators let us combine conditions:

#   &  AND  (both must be TRUE )
# Is the patient overweight AND a smoker?
# BMI cutoff = 25
(BMI > 25) & (smoking_status == "Yes")
(BMI < 25) & (smoking_status == " Yes")


#   |  OR   ( at least one must be TRUE )

# Is the patient is overweight OR a smoker
(BMI > 25) | ( smoking_status == "Yes")
BMI
smoking_status

#   !  Not  (reverse the condition)

# Is the patient Not a smoker
! smoking_status == "No"
# condition = yes
# output = FALSE







# -------------------------
# 2. Data structures in R
# -------------------------
# Data structures are how R organizes and stores information.

# Main structures we will use again and again:
#  1. vectors
#  2. Lists
#  3. Matrices
```

```r
#  4. Data Frames

# --------------
# vectors
# -------------
# simplest data structure in R.
# It stores a sequence of values, but all of the same type.

#    - Numeric vector
num_vec <- c(1,2,3,4)
class(num_vec)

# numeric vectors used to perform mathematical calculation

# - character vector
chrc_vector <- c("gene1","gene2","gene3")

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

mix_vector <- c("gene1", "gene2", 2)
mean(mix_vector)


# We can extract values from vectors using indexing with [].
num_vec[2]
num_vec[2:4] # : indicates sequence

# you can only combine vectors of equal sequence

# We can treat vectors as column as row

# by column
vec_col <- cbind(num_vec,chrc_vector)

# ------------
# Lists
# -------------
# Unlike vectors, a list can hold multiple types together:
# numbers, text, logical even other data frames.

all_vectors <- list(num_vec, chrc_vector, logical_vector)

# save your raw_data
# save processed data
# results

# We access elements with [[ ]]
all_vectors [[2]]







# ---------------
# Matrices
# ---------------
# A matrix is a 2D structure (rows x columns).
# All values must be the same type (usually numeric).
# Example: gene expression matrix where rows are genes and
# columns are samples.


my_matrix <- matrix(1:9, nrow = 3, ncol = 3)
```

```r
# By default R fills the matrix column wise
# we change using byrow = TRUE
my_matrix <- matrix(1:9, nrow = 3, nicol =3, byrow = TRUE)



# We access elements with [row, column]
my_matrix[2,3]
my_matrix [2,# all col]
my_matrix [#rows, #columns]




# -------------
# Data Frames
# -------------
# A data frame is the most important structure for real datasets
# Each column can be of a different type: numeric, character, factor
data <- data.frame(
  patient_id = c("P1", "P2", "P3"),
  age = c("cancer","diabetes", "cancer")
)

print(data)



# --------------------
# Dataset Assessment
# --------------------

# Functions like str(), head(), dim(), and names() help us explore
# the dataset before analysis.
str(data) # structure of the dataset
head(data) # first 6 rows
head(data, n=2 ) # first 6 rows
tail(data) # last 6 rows
tail(data, n =2)
dim(data) # rows & column
names (data) # column names

# Data frame are indexed like matrix with flexibility
# access a column directly
data$patient_id
data[c(1,3), c(2,3)]







# ---------------
# Missing Values
# ---------------
# Real data is messy. Missing values in R are written as NA.
# They can ruin calculations if not handled

# We can:
#   - detect them: is.na()
is.na(data)

#   - count them: sum(is.na())
```

```r
sum(is.na(data))

#   - missing values by column
colSum(is.na(data))

#   - missing values by rows
rowSum(is.na(data))


#   - remove them: na.omit()
# remove rows with missing values
clean_data_1 <- na.omit(data)
clean_data_1

#   - remove column with missing value
clean_data_2 <- data[, colsums(is.na(data))==0]

#   - replace them: impute with 0 or column mean
clean_data_3 <- data

clean_data_3[is.na(clean_data_3)] <- 0

clean_data_4 <- data
clean_data_4[is.na(clean_data_4)] <- mean (data$age, na.rm = TRUE)
clean_data_4


# ---------------------------------
# 3. Functions in R
# ---------------------------------
# Functions let us wrap code into reusable blocks.

# function is reusable block of

# Structure of a function:
#   - name
# Calculate_BMI

#   - arguments (inputs )
function(x) # input argument

#   - body (steps/operation)
 [Bmi <- weight/(height^2)]

#   - return value (output)
return(Bmi)



# why use functions?
#   - Avoid repetition
#   - Organize and simplify code
#   - Reuse across projects
#   - Share with others
#


# Create a function to calculate BMI
# first argument = weight
# second argument = height
calculate_BMI <- function(weight, height){
  # operation we want perform
  bmi <- weight/(height^2)

  return(bmi)
```

```
}

# call the function
calculate_BMI(weight = 60, height = 1.75)
calculate_BMI(weight = weight, height = height)
calculate_BMI(60)
# if your function is expecting two arguments

calculate_BMI <- function(weight, height, age) {
  # operation we want to perform
  bmi <- weight/(height^2)



  return(bmi)
}

calculate_BMI(60,1.65)



# -------------
# Summary:
# -------------
# Functions help us package logic once and it to different inputs

# ----------------------------------
# 4. Automating workflows with loop
# ----------------------------------
# Suppose you have multiple datasets and you want to:
#   - import them,
#   - check missing values,
#   - clean columns,
#   - compute BMI,
#   - and save results
#
# Instead of repeating steps for each file, we use loops.
#
# Typical loop workflow:
#  1. Define input and output folders
input_dir <- "Raw_Data"
output_dir <- "Results"

# create output folder if not already exsist
if(!sir.exists(output_dir)){
  dir.create(output_dir)
}
# the Results folder if we be created if already not exsist



#  2. List which files to process
files_to_process <- c("data_1.csv","data_2.csv") # names of files


#  3. For each file:
result_list <- list()


#        - import data
#        - handle NA values
#        - calculate BMI
#        - save results (both as CVS and inside R list)
#
```

```
for (file_names in files_to_process){
  cat("\nProcessing:", file_names,"\n")

  input_file_path <- file.path(input_dir, file_names)

  # Import dataset
  data <- read.csv(input_file_path, header = TRUE)
  cat("File imported. Checkinh for missing values...\n")

  # handling missing values

  if("height" %in% names(data)){
    missing_count <- sum(is.na(data$height))

    cat("Missing values in 'height:", missing_count, "\n")
    data$height[is.na(data$height)] <- mean(data$height, na.rm = TRUE)
  }

  if("weight %in%" names(data)){
    cat("Missing values in 'weight':",missing_count, "\n")
    data$weight[is.na(data$weight)] <- mean(data$weight, na.rm = TRUE)

  }
  # calculate BMI
  data$bmi <- calculate_BMI(data$weight, data$height)
  cat("BMI has been calculated successfully.\n")

  # save results in R
  result_list[[file_names]]  <- data

  # save results in Results folder
  output_file_path <- file.path(output_dir, paste0("BMI_results",file_names))
  write.csv(data, output_file_path, row.names = FALSE)
  cat("Res9daved to:", output_file_path, "\n")
}
```