

Calculus II

With R

“Statistics is the grammar of science”

Karl Pearson

Course Objectives

- Familiarity with R Programming Basic
- Numerical Calculations in R
- Plotting Mathematical Graphs
- Working with Data and Data Structures in R
- Utilizing Advanced R Packages
- Solving Practical Problems Using R



Grading Breakdown

Total Points: 5 / 20

- 2 Points for Assignments
- 1 Points for Project
- 2 Point for Final Quiz
- 0.5 Extra Point for Attendance and 0.5 Extra Point for Class Participation

Questions related to the projects will be asked at the end of the term. Students will receive their project scores based on their understanding and mastery of the project content.

Course Resources

You can access to course resources including docs, exercises etc... via Github.

<https://github.com/adelamdi/Calculus2-with-R>

- Any contribution to GitHub will count as extra points for students, including:
 1. Sharing advanced R codes related to the topics
 2. Providing creative solutions to complex problems
 3. Improving or refining existing codes
- These activities will be considered for bonus points.
- Quera?

Why R?

**Extensive
Statistical
Libraries**

**Free and Open
Source**

**User-Friendly
Interface**

**Powerful
Statistical
Capabilities**

Educational Use

**Strong
Community
Support**



Installing R and RStudio

1. Download R

Visit the Comprehensive R Archive Network (CRAN) at [CRAN R Project](https://cran.r-project.org/) to download the latest version of R for your operating system (Windows, macOS, or Linux).

2. Installation Process

Follow the installation instructions specific to your operating system.

Accept the default settings for a smooth installation process.

3. Install RStudio (Optional but Recommended)

Download RStudio from [RStudio Website](https://www.rstudio.com/) for a more user-friendly interface to work with R.

Follow the installation instructions for RStudio.

Introduction to RStudio Environment

1. What is R Studio?
2. Console
3. Script Editor
4. Environment/History Pane
5. Files/Plots/Packages/Help Pane

Using Google Colab

1. Introduction to Google Colab

Google Colab is a free, cloud-based Jupyter notebook environment that allows you to write and execute Python and Rcode in your browser

2. Accessing Google Colab

Go to Google Colab

Sign in with your Google account to access the platform

3. Creating a New Notebook

Click on "File" > "New Notebook" to create a new notebook.

Advantages of R Studio

Here are some key advantages of using RStudio:

- Powerful and user-friendly IDE for writing, reading and debugging R code
- Easy to use project management tool.
- Data visualization support
- Code output organization
- Markdown reading and writing etc...

Advantages of Google Colab

Here are some advantages of using Google colab:

- Cloud-based and free
- Access to GPU and TPU
- Support for R + Python
- Save and access files via Google drive
- Support for a wide range of ready to use libraries



Lets Begin!

Writing Our First Program in R

Program 1

number1 <- 10 # Assign value to the first number

number2 <- 5 # Assign value to the second number

sum <- number1 + number2 # Calculate the sum

print(sum) # Print the result

Program 2

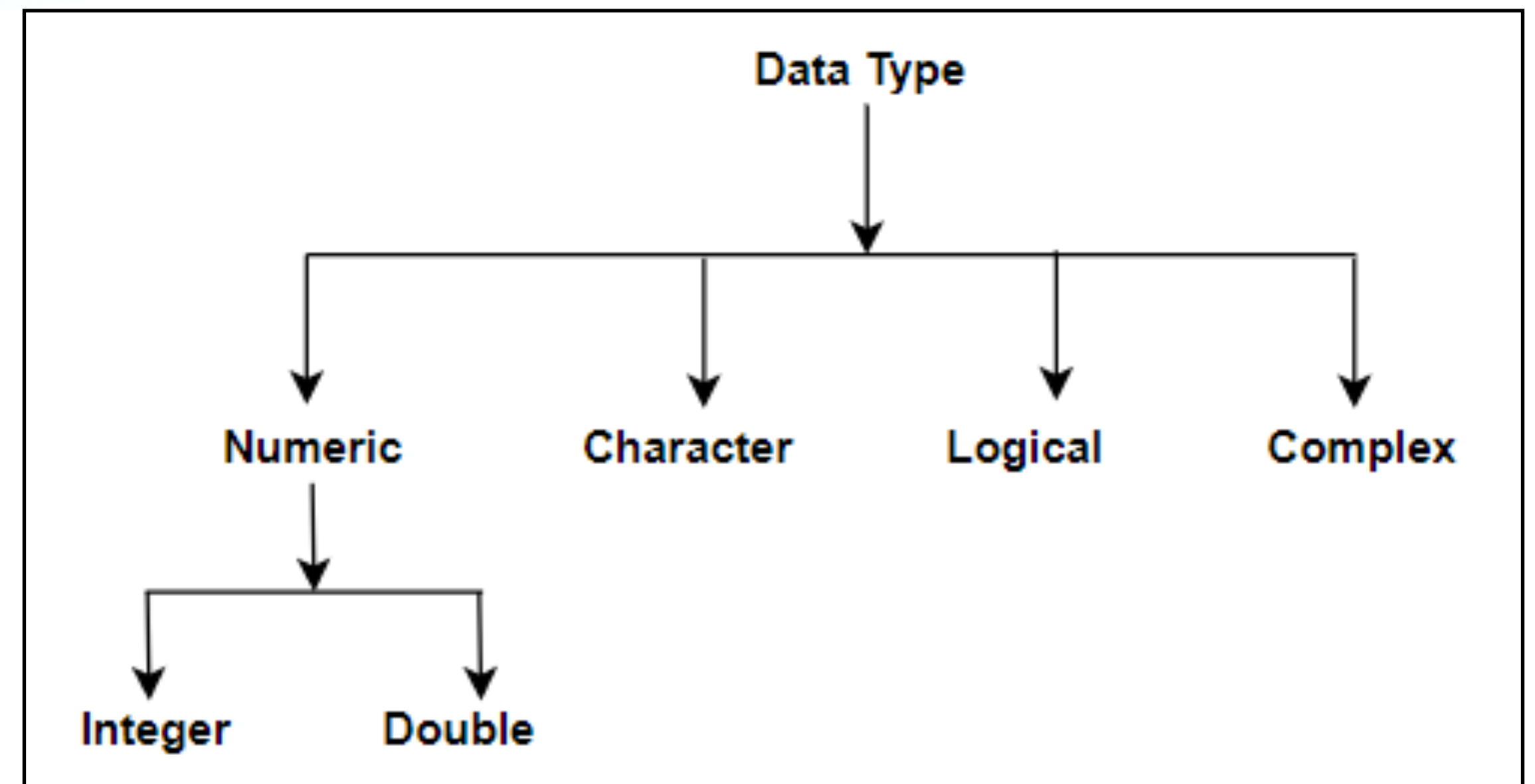
plot(1:10)

Variable Types

- Numeric
 - Integer
 - Character (String)
 - Logical (Boolean)
 - Complex
- Class Function - `class()`

Type Conversion

- `as.numeric()`
- `as.integer()`
- `as.complex()`



Operators

R divides the operators in below groups:

- Assignment operators
- Arithmetic operators
- Comparison operators
- Logical operators
- Miscellaneous operators

Assignment Operators

- Local assignment (->)(<-)
- Global assignment (->>)(<<-)

```
txt <- "global variable"
my_function <- function() {
  txt = "fantastic"
  paste("R is", txt)
}
my_function()
txt # print txt
```

- Whats the difference?

Arithmetic Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponent (^)
- Modulus (%)
- Integer Division (%/%)

Comparison Operators

- Equal (==)
- Not equal (!=)
- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)

Logical Operators

- Element-wise logical AND (&)
 - Logical AND (&&)
 - Element-wise logical OR (|)
 - Logical OR (||)
 - Logical not (!)
-
- & and | are vectorised.
 - && and || are short-circuited. (Right side of logical operator will be evaluated only if left side doesn't determine the outcome)

Miscellaneous Operators

- Creates a sequence of numbers (:)
- Find out if an element belongs to a vector (%in%)
- Matrix multiplication (%*%)

Getting Input from User

- We can use `readline(prompt = "Your message here")` to get input from the user.
- It returns the input as a string, so you may need to convert it to numeric or other types.

Example: Getting two numbers from the user

```
x <- as.numeric(readline(prompt = "Enter the value of x: "))
```

```
y <- as.numeric(readline(prompt = "Enter the value of y: "))
```

Print the values

```
cat("You entered:", "x =", x, "and y =", y, "\n")
```

Conditions (If Statements)

- If - else if - else

```
a <- 200
b <- 33
if (b > a) {
  print("b is greater than a")
} else if (a == b) {
  print("a and b are equal")
} else {
  print("a is greater than b")
}
```

- Nested if
- Conditions + Logical operators

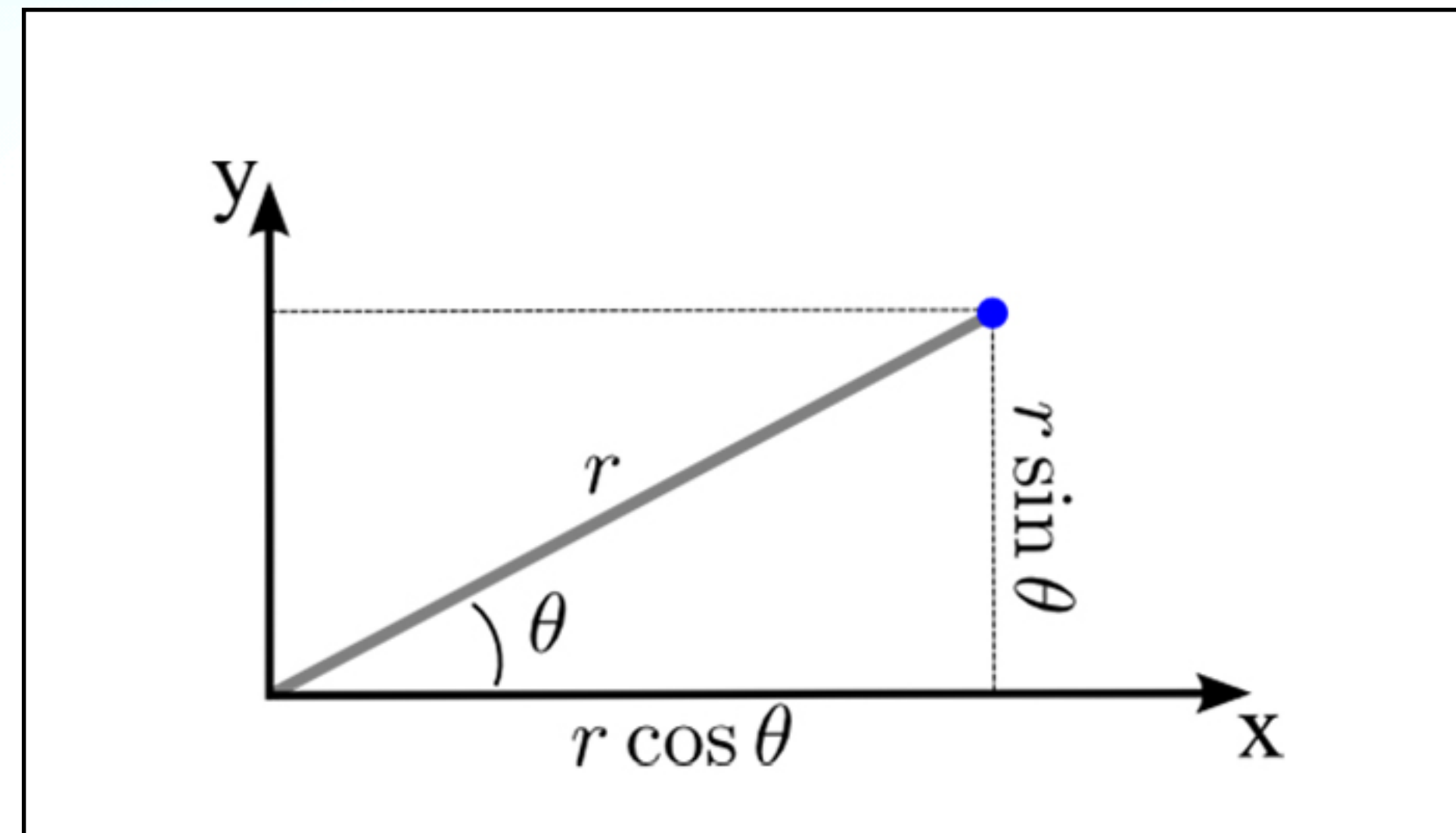
Simple Math

We can use arithmetic operators to perform common basic mathematical operations of numbers. There are also some built-in math functions like:

- `min()` and `max()`
- `sqrt()`
- `abs()`
- `ceiling()` and `floor()`
- `sin()` and `cos()`

Practical Challenge

- Write an R code to convert polar coordinates to cartesian and vice versa.



Loops

Loops can execute a block of code as long as a condition is satisfied.

R has two loop commands:

- For loop
- While loop

For Loop

For loop is used for iterating over a sequence. With for loops we can execute a block of code, once for each item in a vector, sequence, array, list and etc...

```
# First Loop  
for (x in 1:10) {  
  print(x)  
}
```

```
# Second Loop  
companies <- list("apple", "samsung", "lg")  
for (x in companies) {  
  print(x)  
}
```


While Loop

While loop will execute a block of code as long as a condition is true.

```
i <- 0
while (i < 6) {
  i <- i + 1
  if (i == 3) {
    next
  }
  print(i)
}
```

- What is “next”?

R Functions

A function is a reusable block of code that perform a specified task. Functions in R allow us to organize our code, avoid code repetition and make modular projects.

Functions are defined using `function()` keyword and can take arguments or inputs and return a value or output.

```
function_name <- function(arg1, arg2, ...) {  
  # Function Codes  
  result <- some_operation(arg1, arg2)  
  return(result)  
}  
function_name
```

- What are arguments?
- What are return items?

A Simple Function

Define a function that calculates the square of a number

```
square <- function(x) {  
  result <- x^2 # Perform the operation  
  return(result) # Return the result  
}
```

Call the function with an argument

```
print(square(5))
```

Lets Solve This Challenges!

1. Write two functions to convert radian to degree and vice versa.
2. Write a function to calculate points from input x and y on below parametric curve. (Variable t is parameter and r = 5 is the radius)

$$x = r . \cos(t)$$

$$y = r . \sin(t)$$

Recursion

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem.

Every recursion has two part:

1. Base case: this part is used to stop recursion from running.
 2. Recursive case: where the function calls itself.
- Example, calculating the factorial of a number

```
factorial <- function(n) {  
  if (n == 1) {  
    return(1) # Base case  
  } else {  
    return(n * factorial(n - 1)) # Recursive case  
  }  
}
```

Recursion Applications

Recursion can be useful for problems like:

- Exponentiation

```
power <- function(base, exp) {  
  if (exp == 0) {  
    return(1) # Base case  
  } else {  
    return(base * power(base, exp - 1)) # Recursive call  
  }  
}
```

```
power(2, 3) # Output: 8
```


Recursion Applications

- Fibonacci sequences

```
fibonacci <- function(n) {  
  if (n == 0) {  
    return(0) # Base case  
  } else if (n == 1) {  
    return(1) # Base case  
  } else {  
    return(fibonacci(n - 1) + fibonacci(n - 2)) # Recursive call  
  }  
}
```

```
fibonacci(6) # Output: 8
```

Vectors

A vector contains a list of items with the same type separated by a comma. We use `c()` keyword to declare vectors.

```
# Example One  
# Vector of strings  
fruits <- c("banana", "apple", "orange")  
# Print fruits  
fruits
```

```
# Example Two  
values = c(TRUE,FALSE,FALSE)  
# Print values  
values
```


Access and Modify Vectors

- We can access a vector item by its index number inside brackets. Indexes start from 1,2,...,n.

```
fruits <- c("banana", "apple", "mango")
```

```
# Accessing the first item
```

```
fruits[1]
```

- We also can change a vector item by its index.

```
fruits <- c("banana", "apple", "orange")
```

```
# Change the first item (banana)
```

```
fruits[1] = "Mango"
```

```
fruits
```

Sort Vectors

- We can use `sort()` function to sort items alphabetically or numerically.

```
fruits <- c("banana", "apple")  
numbers <- c(2, 3, 51, 1, 20, 4)
```

```
sort(fruits) # Sort a string  
sort(numbers) # Sort numbers
```


Lists

- Lists in R are like vectors but we can store many different data types in lists.

```
# List of strings
```

```
thislist <- list("apple", "banana", "cherry", 2, TRUE)
```

```
# Print the list
```

```
thislist
```

List Functions

- Access a specific item and change it

```
ls1 <- list("apple", "banana", "cherry")
```

```
ls1[1]
```

```
ls1[1] <- "blackcurrant"
```

```
ls1[1]
```

```
# Print the updated list
```

```
ls1
```

- Add item to list

```
ls1 <- list("apple", "banana", "cherry")
```

```
ls1
```

```
ls1 = append(ls1, 'mango')
```

```
# Print the updated list
```

```
ls1
```


List Functions

- Check if item exist in list

```
ls1 <- list("apple", "banana", "cherry", "mango")  
if ("mango" %in% ls1){  
  print('There is mango')  
}
```

- Remove from list with index or value

```
ls1 <- list("apple", "banana", "cherry", "mango")  
# Delete with value  
ls2 <- ls1[ls1 != "mango"]  
# Delete with index  
ls3 <- ls1[-4]  
# Print modified lists  
ls2  
ls3  
}
```

List Functions

- Loop through a list

```
ls1 = list("BMW","Mercedes","Hyundai","Lexus")  
for(car in ls1){  
  print(paste('Car:',car))  
}
```

- Join lists

```
ls1 = list("BMW","Mercedes","Hyundai","Lexus")  
ls2 = list("Ninja","Honda")  
ls3 = c(ls1,ls2)  
# Print Joined list  
ls3
```


Matrices

Matrix is a two dimensional data with rows and columns. We can use `matrix()` keyword to create a matrix and specify number of rows and columns with `nrow` and `ncol`.

```
m1 = matrix(c(1,2,3,4,5,6,7,8,9),nrow = 3,ncol = 3,byrow = TRUE)
# Printing the matrix
m1
```

- What is “byrow=TRUE”?

Matrix Functions

- Access and change values

```
m1 = matrix(c(1,2,3,4,5,6,7,8,9),nrow = 3,ncol = 3,byrow = TRUE)
```

```
# Changing row 1, col 2 index
```

```
m1[1,2] = 10
```

```
m1
```

- Add rows and columns

```
m1 = matrix(c(1,2,3,4),nrow = 2,ncol = 2,byrow = TRUE)
```

```
# Add a column
```

```
m2 <- cbind(m1, c("cbind1", "cbind2"))
```

```
# Add a row
```

```
m3 <- rbind(m1,c("rbind1","rbind2"))
```

```
# Printing new matrixes
```

```
m2
```

```
m3
```


Matrix Functions

- Remove rows and columns

```
m1 = matrix(c(1,2,3,4,5,6),nrow = 2,ncol = 3,byrow = TRUE)
```

```
m2 <- m1[-c(1),] # Remove first row
```

```
m3 <- m1[, -c(1)] # Remove first column
```

```
# Print modified matrices
```

```
m2
```

```
m3
```

- Loop through a matrix

```
m1 = matrix(c(1,2,3,4,5,6),nrow = 2,ncol = 3,byrow = TRUE)
```

```
for(row in 1:nrow(m1)){
```

```
  for(col in 1:ncol(m1)){
```

```
    print(m1[row,col])
```

```
  }
```

```
}
```

Arrays

Unlike matrices, arrays can have more than 2 dimensions in R. We use `array()` function to declare an array and `dim` keyword to determine array dimensions. Remember that array can only store data with same data types.

```
# An array with more than one dimension created from a vector at first
```

```
vector1 = c(1:36)
```

```
arr1 <- array(vector1, dim = c(4, 3, 3))
```

```
arr1
```

- How can we access and modify array items?

DataFrames

- Data frames are data displayed in a format like table. We can store data with different data types in a data frame but each column in data frame can contain data with same data type. We use `data.frame()` keyword to declare a data frame.

```
# Create a data frame
```

```
df1 <- data.frame (  
  ID = c(1, 2, 3),  
  Name = c("Alex", "John", "Alice"),  
  Age = c(60, 30, 45)  
)
```

```
# Print the data frame
```

```
df1
```

- What does `summary()` keyword do?

DataFrame Functions

- Access and modify values

```
# Create a data frame
```

```
df1 <- data.frame (  
  ID = c(1, 2, 3),  
  Name = c("Alex", "John", "Alice"),  
  Age = c(60, 30, 45)  
)
```

```
# Access to values
```

```
df1[2]
```

```
df1[[2]]
```

```
df1$Name
```

```
# Modify a value
```

```
df1$Name[2] = "Richard"
```

```
df1
```

Other functions like add and remove rows and columns, combine data frames and etc.. are like arrays function.

Plotting

We can use `plot()` function to draw points in a diagram. This function has two main arguments, x-axis and y-axis. For example, you can plot two numbers (1,4) and (2,9) with code below:

```
plot(c(1, 2), c(4, 9))
```

- Plot a simple sequence

```
plot(1:10)
```

- Drawing a line

```
plot(1:10, type="l")
```

- Labeling the plot

```
plot(1:10, main="Graph 1", xlab="X-axis", ylab="Y-axis")
```

Next Subjects

- Introducing R packages
- Ggplot2 for plotting mathematical curves
- Statistics in R (Max, Min, Percentiles, Median, Mean and)
- Numerical integration - integrate() function
- Numerical differentiation - Derive package - 3ways to calculate derivative (pracma package - directly use h and definition of derivative - mathematical solving and defining a new function
- Series and sequences
- Multivariable function
- Partial derivatives
- Gradient, divergence, curl
- Parametric curves arc length
- Multivariate functions maxima and minima