R programming language

history

- Robert Gentleman has developed R in 1993
- Robert got inspirations from another language S [Statistics] from IBM
- developed for performing the statistical calculations

features

- it is not a general purpose development language
- can not be used for
    - developing console applications
    - developing web applications (*)
    - developing mobile applications
- can be used to develop applications involving the statistical calculations
- provides the general purpose programming constructs
    - variable declarations
    - function declaration and invocation
    - loops
    - condition
    - data types
    - collections
- provides in-built statistical function
- provides various types of packages
    - graphical packages (ggplot)
    - machine learning packages
    - data analysis packages
    - data analytics packages
- is a scripting language (there is no entry point function)

environment setup

- install latest version of R programming language from

  https://cran.r-project.org/bin/windows/base/

- IDEs

    - RStudio [https://rstudio.com/products/rstudio/download/]
    - Pycharm [https://www.jetbrains.com/help/pycharm/r-plugin-support.html]

- Editor

    - VSCode

- update system environment variable named path with the bin directory of python and R

execute the code

- using command prompt
    - execute the command R to get the interactive R command prompt
    - write and execute the R code line by line
- using file (script)
    - write the code in a file with an extension R
    - use Rscript to execute the file
    - ```
      > Rscript test.R
      ```

shortcuts in RStudio

- ctrl + l => clear the console
- ctrl + shift + n => creates a new R script
- ctrl + enter => run the current line or selected lines
- ctrl + shift + s => runs the current entire document
- ? => shows help of the function name
- ?? => search for the topic

identifier

- used to identify an entity like variable or a function
- rules to declare identifier
    - can not start an identifier with a number
        - e.g. 1name is invalid
    - can not start an identifier with an underscore
        - e.g. _varname is invalid
    - can not use special characters like space, symbols (#$%@)
        - e.g.
            - invalid identifier (has space within it): first name
            - invalid identifier: name@1, name#, name%
    - identifier can use a special character dot (.)
        - e.g.
            - valid identifier: first.name
            - valid identifier: can.vote
- convention
    - to create an identifier for first name
        - first_name (preferred in python)
        - first.name (preferred in R)

variable

- a memory used to store a value
- a variable can change its value
- variable declaration / assignment
      ```

```r
# python like syntax
num = 100

# single headed arrow syntax
num <- 100

# double headed arrow syntax
num <<- 100

# single headed arrow syntax
100 -> num

# double headed arrow syntax
100 ->> num
```

- to print variable value on console

```r
num = 100
print(num)

# print value of a variable along with a constant string
cat("num = ", num)
paste("num = ", num)
paste0("num = ", num)
```

pre-defined values

- NA
    - Not Available
- Inf
    - stands for infinity

data types

- in R, data type is always inferred
- automatically decided by R by looking at the current value inside the variable
- types
    - Vectors
    - Lists
    - Matrices
    - Arrays

- Factors
- Data Frames

Vectors

- represents one dimensional array
- collection of similar values
- data types
  - **numeric**
    - represents a number
    - e.g.

```
# Vector of numeric value
num = 100

# Vector of numeric value
salary = 10.15
```

  - **integer**
    - represents a whole number
    - e.g.

```
# vector of integer
age = 30L
```

  - **character**
    - represents a string
    - e.g.

```
# vector of character
first.name = "steve"

# vector of character
last.name = 'jobs'
```

  - **logical**
    - represents boolean values like

        - TRUE or FALSE
        - T or F
    - e.g.

```
# vector of logical
can.vote = TRUE
```

- **raw**
    - represents raw characters
    - e.g.

```
# vector of raw
address = charToRaw('pune')

# vector of raw
hello = charToRaw('नमस्कार')
```

- **complex**
    - represents a complex number
    - complex number contains
        - real part
        - imaginary part
    - e.g.

```
# vector of complex
complex.number = 10 + 20i
```

- a collection of similar values

```
# vector of numeric
# v.1 = [10, 20, 30, 40, 50, 1]
v.1 = c(10, 20, 30, 40, 50, TRUE)

# vector of character
# v.2 = ["10", "20", "steve", "30", "40"]
v.2 = c(10, 20, "steve", 30, 40)

# vector of character
```

```
# v.3 = ["10", "20", "30", "40", "50", "TRUE", "india"]
v.3 = c(10, 20, 30, 40, 50, TRUE, "india")
```

- **indexing**
  - index starts at one (1) [does not start at zero (0)]
  - positivie indexing
    - e.g.

```
numbers = c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

# get the first value from the vector
# 10
print(numbers[1])

# get the last value from the vector
# 100
print(numbers[10])

# get the values at postions 2, 4 and 6
# [20, 40, 60]
print(numbers[c(2, 4, 6)])
```

  - negative indexing
    - exclude the index and return remaining values
    - e.g.

```
numbers = c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

# get all the values except first position
# [20, 30, 40, 50, 60, 70, 80, 90, 100]
print(numbers[-1])

# get all the values except last position
# [10, 20, 30, 40, 50, 60, 70, 80, 90]
print(numbers[-10])

# get all the values expect from positions 2 4 and 6
# [10 30 50 70 80 90 100]
print(numbers[c(-2, -4, -6)])
```

- **slicing**

- taking a portion out of a vector
- e.g.

```
numbers = c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

# 60, 70, 80, 90, 100
print(numbers[-1:-5])

# 60, 70, 80, 90, 100
print(numbers[6:10])
```

- **broadcast operator**
  - performing operation on every member of the vector
  - **mathematical operators 1**

```
# add 10 to every number
print(numbers.1 + 10)

# subtract 10 from every number
print(numbers.1 - 10)

# multiply 10 to every number
print(numbers.1 * 10)

# divide every number by 2
print(numbers.1 / 2)
```

  - **mathematical operators 2**

```
# perform addition on every member of numbers.1 and numbers.2
print(numbers.1 + numbers.2)

# perform subtraction on every member of numbers.1 and numbers.2
print(numbers.1 - numbers.2)

# perform division on every member of numbers.1 and numbers.2
print(numbers.1 / numbers.2)

# perform multiplication on every member of numbers.1 and
numbers.2
print(numbers.1 * numbers.2)
```

- **logical operators**

```
numbers.3 = c(TRUE, TRUE, TRUE, FALSE, FALSE, TRUE, FALSE)
numbers.4 = c(FALSE, TRUE, TRUE, FALSE, FALSE, FALSE, TRUE)

# broadcast
# FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
print(!numbers.3)


# non-broadcast operators

# TRUE
print(numbers.3 && TRUE)

# TRUE
print(numbers.4 || TRUE)


# broadcast operators

# TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
print(numbers.3 & TRUE)

# TRUE TRUE TRUE TRUE TRUE TRUE TRUE
print(numbers.4 | TRUE)
```

Lists

- multi-dimensional collection
- collection made up with multiple vectors
- to create a list call a function list()
- e.g.
  - list created without temporary names

```
list.1 = list(
  c(10, 20, 30, 40, 50),
  c(60, 70, 80, 90, 100)
)

# [[1]]
# [1] 10, 20, 30, 40, 50
# [[2]]
# [2] 60, 70, 80, 90, 100
```

```
# 10, 20, 30, 40, 50
print(list.1[[1]])

# 20
print(list.1[[1]][2])

# 60, 70, 80, 90, 100
print(list.1[[2]])

# 90
print(list.1[[2]][4])
```

- list created with temporary names

```
persons = list(
  personNames = c("person1", "person2"),
  personAddresses = c("pune", "mumbai")
)

# $personNames
# [1] "person1" "person2"
# #personAddresses
# [1] "pune" "mumbai"

# "person1" "person2"
print(persons$personNames)

# person2
print(persons$personNames[2])

# "pune" "mumbai"
print(persons$personAddresses)

# pune
print(persons$personAddresses[1])
```

Matrices

- two dimensional collection
- to create a matrix call a function named matrix()
- e.g.

```
m.1 = matrix(c(10, 20, 30, 40), nrow = 2, ncol = 2)
```

```
#      [, 1]   [, 2]
# [1, ]    10      30
# [2, ]    20      40


m.2 = matrix(c(10, 20, 30, 40), nrow = 2, ncol = 2, byrow = TRUE)

#      [, 1]   [, 2]
# [1, ]    10      20
# [2, ]    30      40
```

- retrieving values from matrix

```


m.1 = matrix(c(10, 20, 30, 40), nrow = 2, ncol = 2)

#      [, 1]   [, 2]
# [1, ]    10      30
# [2, ]    20      40

# [1] 10 30
print(m.1[1, ])

# [1] 30 40
print(m.1[, 2])

# [1] 30
print(m.1[1, 2])
```

- to make the retrieval easier, matrix provides a naming rows and columns convention
- e.g.

```


rownames = c("car1", "car2")
colnames = c("model", "average", "price")
cars = matrix(c("i20", "i10", 17, 18, 7.5, 5.5),
    nrow = 2, ncol = 3, dimnames = list(rownames, colnames))

#        "model"   "average"   "price"
# "car1"    "i20"         17        7.5
# "car2"    "i10"         18        5.5

# 7.5
print(cars["car1", "price"])

# i10
print(cars["car2", "model"])
```

- recycling values while creating a matrix

```
m.5 = matrix(c(10, 20, 30), nrow = 3, ncol = 2)

#      [,1] [,2]
# [1,]   10   10
# [2,]   20   20
# [3,]   30   30

m.6 = matrix(c(10, 20, 30), nrow = 3, ncol = 2, byrow = TRUE)

#      [,1] [,2]
# [1,]   10   20
# [2,]   30   10
# [3,]   20   30
```

Arrays

Factors

- the way to store categorical data
- the values will get stored as numeric positions of levels
- e.g.

```
factor.1 = factor(c('red', 'green', 'red', 'green', 'green'))

# red green red green green
# Levels: red green

factor.2 = factor(
    c('red', 'green', 'red', 'green', 'green'),
    levels = c('red', 'green'),
    labels = c(1, 2)
)

# 1 2 1 2 2
# Levels: 1 2
```

Data Frames

- tabular representation of the data
- similar to pandas data frame

- collection of rows and columns
- two dimensional collection (Vectors)
- list of lists
- e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com", "p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

#     name age       email address
# 1 person1  10 p1@test.com    pune
# 2 person2  20 p2@test.com  mumbai
# 3 person3  15 p3@test.com  nashik
# 4 person4  18 p4@test.com  satara
```

- **functions**
  - **class**
    - used to get the type of data structure
    - e.g.

```
v = c(10, 20, 30, 40, 50)

# numeric
print (class(v))
```

  - **typeof**
    - used to get the data type of data structure
    - e.g.

```
v = c(10, 20, 30, 40, 50)
```

```
# double
print (typeof(v))
```

- **names**
  - used to set or get the temporary names for every position in the data structure
  - used with data.frames and lists
  - e.g.

```
list.1 = list(names = c("person1", "person2"), ages = c(20, 30))

# names ages
print(names(list.1))
```

- **str**
  - similar to info() in pandas
  - used to get the basic information about the data structure
  - used with data frames and lists
  - the information includes
    - number of observations (rows)
    - number of variables (columns)
    - name of every variable
    - data type of every variable
    - preview of data in every column
  - e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

print(str(df.persons))

# 'data.frame': 4 obs. of  4 variables:
# $ name   : chr  "person1" "person2" "person3" "person4"
# $ age    : num  10 20 15 18
# $ email  : chr  "p1@test.com" "p2@test.com" "p3@test.com"
```

```
"p4@test.com"
# $ address: chr  "pune" "mumbai" "nashik" "satara"
```

- **summary**
  - similar to describe() function in pandas
  - used to get statistical information about the data structure
  - can be used with data frames and lists
  - information includes
    - data type of column
    - for numeric column
      - minimum value
      - maxivalue
      - median
      - mean
      - 1st quartile
      - 3rd quartile
  - e.g.

```
  names = c("person1", "person2", "person3", "person4")
  ages = c(10, 20, 15, 18)
  emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
  addresses = c("pune", "mumbai", "nashik", "satara")

  df.persons = data.frame(
    name = names,
    age = ages,
    email = emails,
    address = addresses
  )

  print(summary(df.persons))

  #           name                age           email
address
  # Length:4           Min.   :10.00   Length:4
Length:4
  # Class :character   1st Qu.:13.75   Class :character   Class
:character
  # Mode  :character   Median :16.50   Mode  :character   Mode
:character
  #                    Mean   :15.75
  #                    3rd Qu.:18.50
  #                    Max.   :20.00
```

- **head**

- used to retrieve first few records
- e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

print(head(df.persons, 2))

#         name age       email address
# 1 person1  10 p1@test.com    pune
# 2 person2  20 p2@test.com  mumbai
```

- **tail**

- used to retrieve last few records
- e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

print(tail(df.persons, 2))

#       name age       email address
# 3 person3  15 p3@test.com  nashik
# 4 person4  18 p4@test.com  satara
```

- **nrow**

- used to get number of rows in a data frame
- e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

# 4
print(nrow(df))
```

- **ncol**
  - used to get number of columns in a data frame
  - e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

# 4
print(ncol(df))
```

- **colnames**
  - used to get column names
  - e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

# "name" "age" "email" "address"
print(colnames(df))
```

- **rownames**
  - used to get row names
  - e.g.

```
names = c("person1", "person2", "person3", "person4")
ages = c(10, 20, 15, 18)
emails = c("p1@test.com", "p2@test.com", "p3@test.com",
"p4@test.com")
addresses = c("pune", "mumbai", "nashik", "satara")

df.persons = data.frame(
  name = names,
  age = ages,
  email = emails,
  address = addresses
)

# "1" "2" "3" "4"
print(rownames(df))
```

Type inspection

- in R type can be inspected by using is functions (functions start with is.*)
- e.g.

```
num = 100
print(is.numeric(num))            # TRUE
```

```
print(is.character(num))          # FALSE

first.name = "steve"
print(is.numeric(first.name))      # FALSE
print(is.character(first.name))    # TRUE
```

Type conversion

- used to convert data from one type to another
- e.g.

```


print(as.numeric("10"))             # 10

print(as.logical("TRUE"))           # TRUE

print(as.character(10))             # "10"
print(as.character(FALSE))          # "FALSE"

print(as.logical(1))                # TRUE
print(as.logical(0))                # FALSE
print(as.logical(100))              # TRUE
print(as.logical(-100))             # TRUE
```

loops

- for loop

```


numbers.1 = c(10, 20, 30, 40, 50)

for (value in numbers.1) {
  print(value)
}
```

functions

- **custom function**
  - user defind functions
  - fucntion can be created with function object
  - syntax

```
# function declaration

# empty function
function1 = function() {
  # function body here
}

# function call
function1()
```

- **built-in fuction**
-
  - **getwd**
    - used to get the current working directory
    - working directory is used to search the files/documents when application wants to read them
  - **setwd**
    - used to set the current working directory
    - working directory is used to search the files/documents when application wants to read them
    - e.g.

```
setwd('/Volumes/data/sunbeam/dataset/data1/')
```

  - **install.package**
    - used to install a package or library and its dependencies
    - e.g.

```
install.package('tidyverse')
```

  - **library**
    - used to load a library in the current environment
    - e.g.

```
library(tidyverse)
```

version numbers

- version number: x.y.z.a
  - x: major version
  - y: minor version
  - z: build version
  - a: nightly build version
- change in the version number
  - major number: breaking changes
  - minor number: enhancement / feature addition / bug fixing
  - build number: bug fixing / nightly build