Project 1: Unit 1

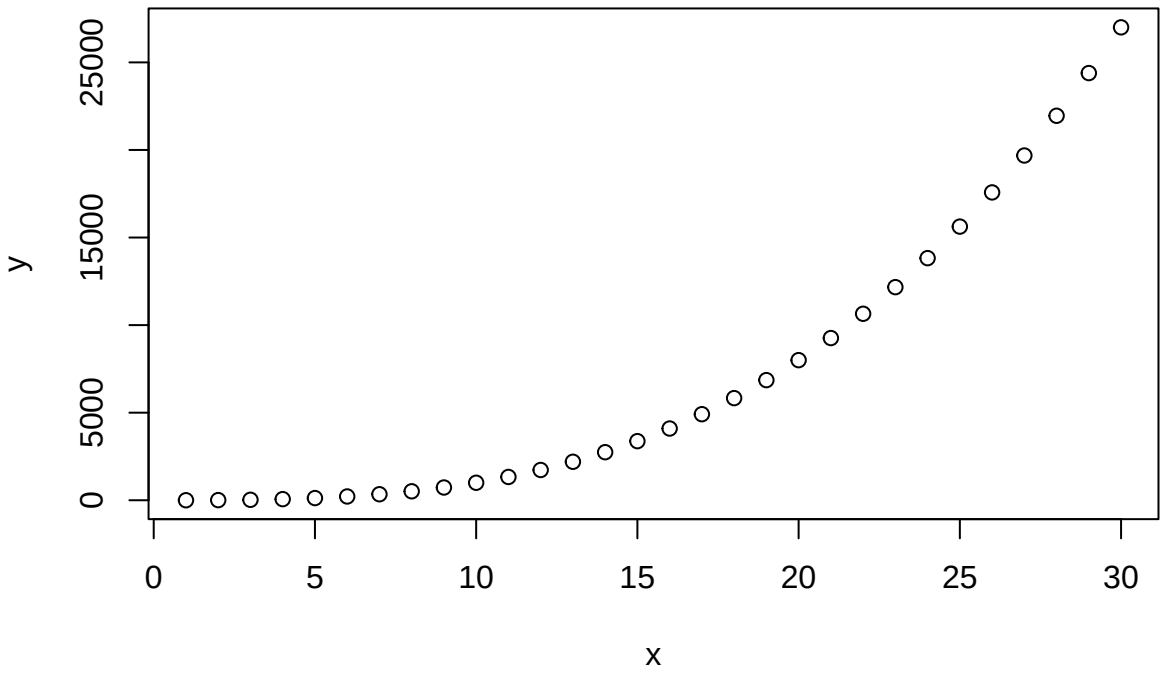
Suman Paudel

3/2/2024

GitHub Repository

Code Execution and Output/Interpretation of Session 4

```
# vector
x <- c(1:30)
y <- x^3
plot(x,y,)
```



Code Sample 1

Interpretation:

- x is vector having elements 1 to 30.
- y is also vector having elements of x exponentiated of 3.
- plot is a generic function in R to plot, by default in R scatter plot is plotted.

```
# store the current working using following command:
initial.dir <- getwd()
initial.dir
```

Code Sample 2

```
## [1] "C:/Users/SumanPaudel/Desktop/R For Data Science"
```

Interpretation:

- **initial.dir** object will be assigned the current working directory.

```
# to change the working directory custom directory
setwd("C:/Users/SumanPaudel/Desktop/R For Data Science")
```

Code Sample 3 ***Interpretation***:

- **setwd()** function will change the working directory to given path.

```
# loading the necessary packages
library(magrittr)
```

Code Sample 4 ***Interpretation***:

- In R, the **library()** function will load and attach the required packages.
- `magrittr` package is loaded.

```r
# to set the output file and bypass the output of R console and R Studio
sink('session4.out')
```

Code Sample 5 ***Interpretation***:

- In R, the **sink()** function will set the output file and bypass the output of R console and R Studio.
- until closed, now all the output will be saved in the session4.out file rather than display on.

```r
# load the dataset from the working directory set earlier
iris <- read.csv('iris.csv')
```

Code Sample 6 ***Interpretation***:

- load the iris dataset from csv file using module **read.csv()** function from current working directory.

```r
# plot the iris dataset to do some analysis
plot(iris)
```

Code Sample 7 ***Interpretation***:

- plot a scatter plot from the iris dataset.
- since we have used **sink()** function earlier now the given output will be saved to session4.out

```r
# summary of iris dataset
summary(iris)
```

Code Sample 8 ***Interpretation***:

- In R, **summary** summary is a generic function used to produce result summaries of the results of various model fitting functions.
- since we have used **sink()** function earlier now the given output will be saved to session4.out

```r
# to close the output file
sink()
```

Code Sample 9 ***Interpretation***:

- Closes the output file that was writing output to it.

```r
# unloading the loaded library
detach("package:magrittr")
```

Code Sample 10 ***Interpretation***:

- Unloads the previous loaded package **magrittr**.
- Usually use to unload or remove the path of R objects, packages which was attached by using **library()**.

```r
# to change back to original directory
setwd(initial.dir)
```

Code Sample 11 ***Interpretation***:

- change back to the directory path which initial.dir object holds.

```r
# load the iris data from UCI machine learning repo (internet archive)
iris <- read.csv(url('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'),header=
```

Code Sample 12 ***Interpretation***:

- fetch the data from online repo using url then convert the obtained into dataframe from it.
- `header = FALSE` will not set the header for dataframe. Often done when we want to explicitly define column names ourselves

```r
# look first few values of data
head(iris)
```

Code Sample 13

```
##    V1  V2  V3  V4          V5
## 1 5.1 3.5 1.4 0.2 Iris-setosa
## 2 4.9 3.0 1.4 0.2 Iris-setosa
## 3 4.7 3.2 1.3 0.2 Iris-setosa
## 4 4.6 3.1 1.5 0.2 Iris-setosa
## 5 5.0 3.6 1.4 0.2 Iris-setosa
## 6 5.4 3.9 1.7 0.4 Iris-setosa
```

Interpretation:

- fetch the first few records from the iris dataframe
- since in previous line of code we did `header = FALSE` our column name is defined as V1, V2, V3, V4, V5.

```r
# add column names for V1, V2, V3, V4, V5 columns to iris dataframe
names(iris) <- c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width", "Species")
```

Code Sample 14 ***Interpretation***:

- set the name of columns in the iris dataframe `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`, and `Species`

```r
# saving the downloaded file to csv in local system
# write.csv(dataframe_name, "path\file_name.csv", row.names=FALSE)
write.csv(iris,'iris.csv')
```

Code Sample 15 ***Interpretation***:

- writes the csv file in the current working directory as `iris.csv` from iris dataframe
- Also we can set the to another path like this `write.csv(dataframe_name,filepath)`
- for example: `write.csv(iris,'C:/Users/SumanPaudel/Desktop/suman.csv')`

```r
# check working directory using
getwd()
```

Code Sample 16

```
## [1] "C:/Users/SumanPaudel/Desktop/R For Data Science"
```

Interpretation:

- returns the current working directory in the console.

```r
# Using forward pipe operator/s in R
# for using pipes in R we need magrittr library
# Hotkey for pipe i.e. is Ctrl + Shift + M in windows and for MacOS CMD + Shift + M
library(magrittr)
iris$Sepal.Length.SQRT <- iris$Sepal.Length %>% sqrt()
```

Code Sample 17 ***Interpretation***:

- computes the square root of `iris$Sepal.Length` and assign it to the new variable `iris$Sepal.Length.SQRT`
- forward pip are very helpful when forwarding and object into function or call the expression

```r
# The assignment pipe, %<>%, is used to update a value by first piping it into one or more rhs expressi
iris$Sepal.Length %<>% sqrt
```

Code Sample 18 ***Interpretation***:

- computes the square root of `iris$Sepal.Length` and assign computed value to itself.
- while using assignment pipe `%<>%`, we must be careful as original data will be lost.

```r
# random seed
set.seed(123)
```

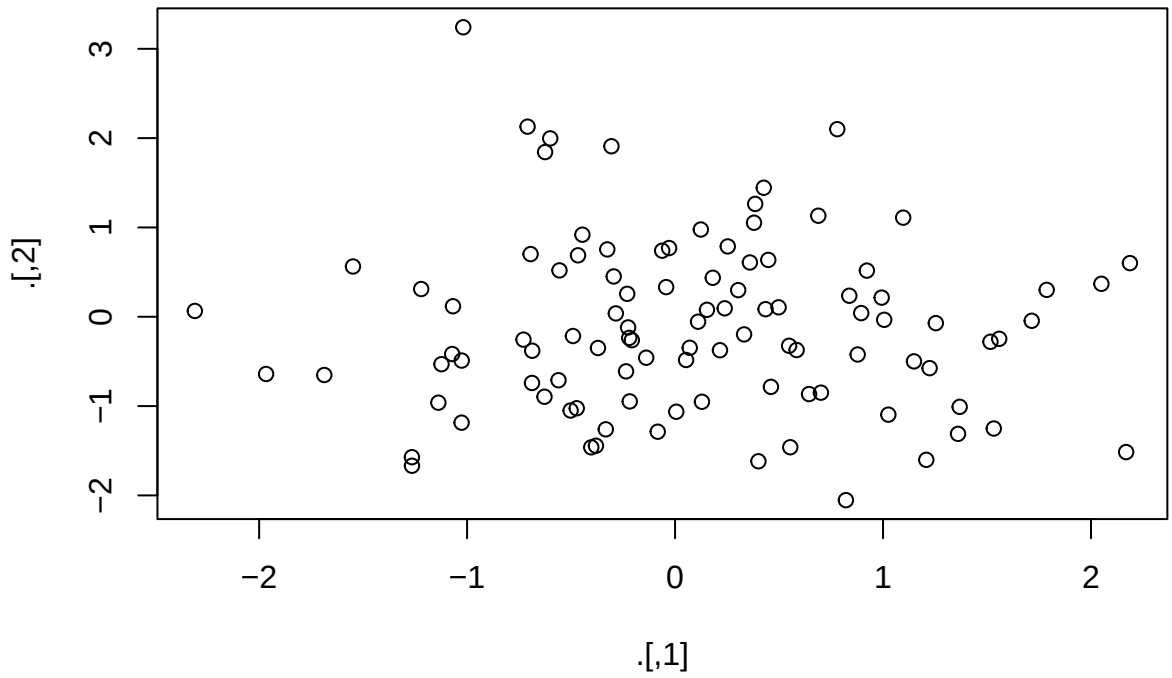
Code Sample 19 *Interpretation*:

- sets random seed to generate random number that can be reproduced
- widely used when dealing with classification problems in machine learning

```r
# rnorm generated the random value from normal distribution
rnorm(200) %>%
  matrix(ncol = 2) %>%
  plot %>%
  colSums
```

Code Sample 20

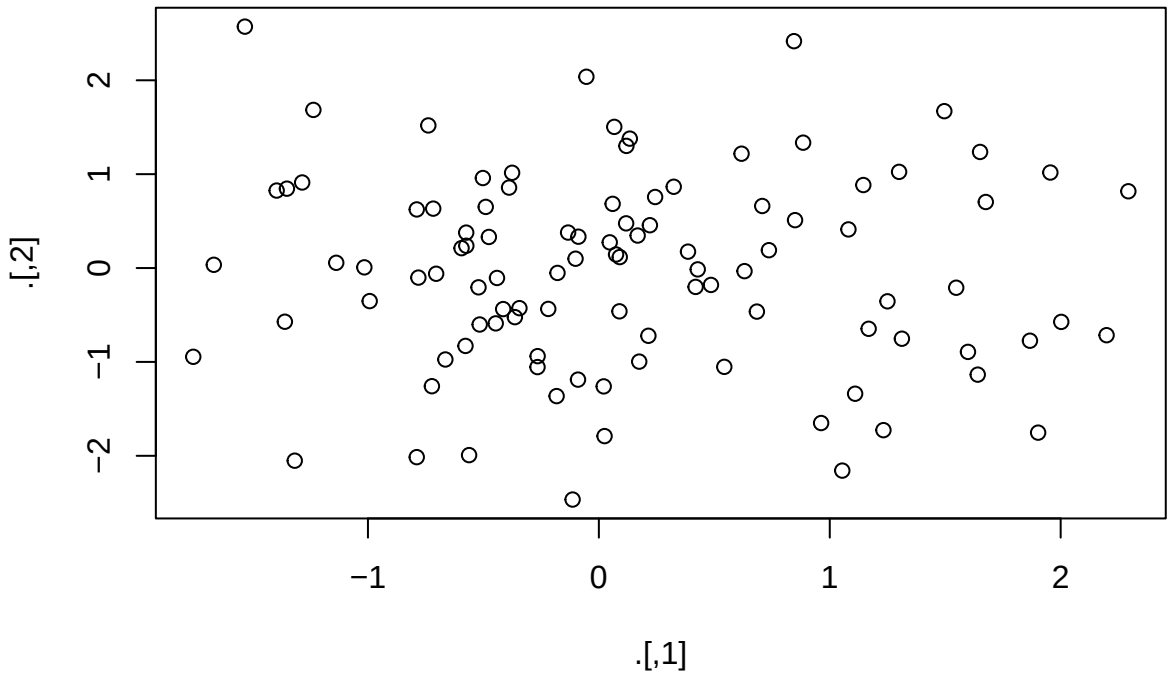
```
## Error in colSums(.): 'x' must be an array of at least two dimensions
```



Interpretation:

- `rnom(200` generates 200 random number from a normal distribution
- then generated values are converted to matrix with 100 * 2 size.
- then plots the values into scatter plot
- when using `colSums` throws an error because code wont get computed after using plot because it won't return anything except the figure or plot.
- To work around this problem, we can use the "tee" pipe.

```
# rnorm generated the random value from normal distribution
rnorm(200) %>%
  matrix(ncol = 2) %T>%
  plot %>%
  colSums
```



Code Sample 21

```
## [1] 12.046511 -3.622291
```

Interpretation:

- `rnom(200` generates 200 random number from a normal distribution
- then generated values are converted to matrix with 100 * 2 size.

- then plots the values into scatter plot
- As we have used `%T%` operator, now both plot and after plot `colSums` also will get computed.

```
# The exposing pipe operator "%$%"
iris %>%
  subset(Sepal.Length > mean(Sepal.Length)) %$%
  cor(Sepal.Length, Sepal.Width)
```

Code Sample 22

```
## [1] 0.3365679
```

Interpretation:

- `%$%` allows us to extract variables from a data frame or list and use them directly in subsequent expressions.
- In the code above, it is used to extract the Sepal.Length and Sepal.Width columns from the iris data frame.
- `iris %>% subset(Sepal.Length > mean(Sepal.Length))` filters the rows of the iris data frame, keeping only those where the Sepal.Length is greater than the mean Sepal.Length. The result is a subset of the original iris data frame.
- `%$% cor(Sepal.Length, Sepal.Width)` after extracting the subset, calculates the correlation coefficient between Sepal.Length and Sepal.Width for which sepal length is greater than it;s mean.

```
cor(iris$Sepal.Length, iris$Sepal.Width)
```

Code Sample 23

```
## [1] -0.114702
```

Interpretation:

- gives the correlation of sepal length and sepal width on the entire iris data frame.

When Not to Use the Pipe

The pipe (`%>%`) is a powerful tool in R, but it's not always the best choice for every situation. Here are some scenarios where we might want to consider alternative approaches:

- If pipes involves more than ten steps, consider breaking it down into smaller chunks with meaningful intermediate objects. This makes debugging easier and improves code readability.
- Pipes work best when transforming a single primary object. If you're dealing with multiple inputs or combining several outputs, the pipe may not be the most suitable tool.
- When your code starts resembling a directed graph with intricate dependencies, using pipes can lead to confusing and convoluted code.
- When developing internal packages pipe might not be the best go to tool.

Final Notes

- `%>%` are used widely in R
- `%<>%` used when assignment of variables are required
- `%T%` and `%$%` used when required explicitly on rare occasion.

Code Execution and Output/Interpretation of Session 5

Naming Convention In R lllowercase - `adjustcolor`
period.separated - `plot.new`
underscore_separated - `numeric_version`
lowerCamelCase - `addTaskCallback`
UpperCamelCase - `SignatureMethod`

The link https://www.r-bloggers.com/2014/07/consistent-naming-conventions-in-r/ suggest to use underscore sperated variable name as it provides better readbility of code.

```
# column vector
round(3.14)
```

Code Sample 1

```
## [1] 3
```

```r
round(3.14, digits = 2)
```

```
## [1] 3.14
```

Interpretation:

- `round(3.14)` rounds the value 3.14.
- `round(3.14, digits = 2)` to nearest 2 digits.

```r
# factorial of 3
factorial(3)
```

```
## [1] 6
```

```r
# factorial of 3*2
factorial(2*3)
```

```
## [1] 720
```

Interpretation:

- `factorial(3)` gives the factorial of 3 which is 6.
- `factorial(3)` gives the factorial of `2*3` which is 720.

```r

# mean of sequence 1:6
mean(1:6)
```

Code Sample 2

```
## [1] 3.5
```

```r
# mean of vector 1 to 30
mean(c(1:30))
```

```
## [1] 15.5
```

Interpretation:

- `mean(1:6)` computes the mean of sequenec 1:6
- `mean(c(1:6)` computes the mean of vector 1:30

```r
# random sampling without and with replacement in R using sample function
# sample size 1 without replacement
die <- 1:6
sample(x = die, size = 1)
```

Code Sample 3

```
## [1] 1
```

```r
# sample size 1 without replacement
sample(x = die, size = 1)
```

```
## [1] 4
```

```r
# sample size 1 with replacement
sample(x = die, size = 1, replace = TRUE)
```

```
## [1] 1
```

Interpretation:

- `sample(x = die, size = 1)` samples the size of 1 from die without replacement. without replacement means each element can only be selected once.
- `sample(x = die, size = 1, replace = TRUE)` samples the size of 1 from die with replacement. with replacement means elements can be selected more than once.

```r
# sample size 2 without replacement
sample(x = die, size = 2)
```

```
## [1] 3 1
```

```r
# sample size 2 without replacement
sample(x = die, size = 2)
```

```
## [1] 3 6
```

```r
# sample size 2 with replacement
sample(x = die, size = 2, replace = TRUE)
```

```
## [1] 2 3
```

Interpretation:

- `sample(x = die, size = 2)` samples the size of 2 from die without replacement. without replacement means each element can only be selected once.
- `sample(x = die, size = 2, replace = TRUE)` samples the size of 2 from die with replacement. with replacement means elements can be selected more than once.

```r
# load the iris dataset from current working directory
iris <- read.csv('iris.csv')
set.seed(123)
```

Code Sample 4 ***Interpretation***:

- `iris <- read.csv('iris.csv')` load the iris dataset from current working directory to iris object variable.
- `set.seed(123)` generates the random number which can be reproduced again. random seed is very useful when in machine learning problems.

```r
# training testing sample
tt.sample <-
  sample(c(TRUE, FALSE),
         nrow(iris),
         replace = T,
         prob = c(0.7, 0.3))
train <- iris[tt.sample,]
test <- iris[!tt.sample,]
```

Code Sample 5 ***Interpretation***:

- `tt.sample <- sample(c(TRUE, FALSE), nrow(iris), replace=T)` samples the value given TRUE or FALSE for given number of rows `nrow` on iris dataset with replacement also with probability weights 0.7 for TRUE and 0.3 for FALSE and store to tt.sample variable.
- train holds traning sample that are subsetted values from tt.sample in iris dataset. Rows corresponding to TRUE in tt.sample are selected
- test holds test sample that are subsetted values which are not TRUE for training set. Rows corresponding to not TRUE in tt.sample are selected
- `!` alters the outcome, like `TRUE` becomes `FALSE`.

```r
# user defined function in R

my_function <- function(){
  ## my function which does nothing
}
```

Code Sample 6 ***Interpretation***:

- R way to define user defined function using `function()`.
- Here, `my_function` is name of the function, `function()` creates the function, inside parenthesis `{}` some expression is evaluated, and last function returns some expression as well.
- A function can take arguments or no arguments it depends upon the situation.

```
# user define function 1 : roll()
# simple function with no argument
roll <- function() {
  die <- 1:6
  dice <- sample(die, size = 2, replace = TRUE)
  cat("Dice Rolled:", dice, '\n')
  sum(dice)
}

# first roll()
cat('First Roll:', roll())
```

Code Sample 7

```
## Dice Rolled: 3 6
## First Roll: 9
```

```
# second roll()
cat('Second Roll:', roll())
```

```
## Dice Rolled: 1 3
## Second Roll: 4
```

```
# third roll()
cat('Third Roll:', roll())
```

```
## Dice Rolled: 5 2
## Third Roll: 7
```

Interpretation:

- The function is named roll with no arguments
- die variable has sequence 1:6 as a die has 6 faces, and sampling is done with sample size two and also with replacement and stored in dice variable.
- the sum function sums the values obtained in dice
- cat just concatenate the values and string just for output.
- After calling the roll function 3 times we can see two dices rolled and their sums.

```r
roll2 <- function(dice = 1:6) {
  dice <- sample(dice, size = 2, replace = TRUE)
  cat("Dice Rolled:", dice, '\n')
  sum(dice)
}

# calling roll2()
cat('First Roll:', roll2())
```

Code Sample 8

```
## Dice Rolled: 6 6
## First Roll: 12
```

Interpretation:

- The function is named roll2 with arguments dice having default value of sequence `1:6`.
- dice variable has default sequence 1:6, and sampling is done with sample size two and also with replacement and stored in dice variable.
- the sum function sums the values obtained in dice
- cat just concatenate the values and string just for output.
- After calling the roll2 function we can see two dices rolled and their sums.

```r
roll3 <- function(dice) {
  dice <- sample(dice, size = 2, replace = TRUE)
  sum(dice)
}

# roll3 function call 1
roll3(dice = 1:6)
```

Code Sample 9

```
## [1] 5
```

```r
# roll3 function call 2. but not correct
roll3(dice = 1:12)
```

```
## [1] 11
```

```r
# roll3 function call 3. but not correct
roll3(dice = 1:24)
```

```
## [1] 40
```

Interpretation:

- The function is named roll3 with arguments dice which is parametric.
- After the input dice is passed into the function, sampling is done with sample size two and also with replacement and stored in dice variable.
- the sum function sums the values obtained in dice
- cat just concatenate the values and string just for output.
- `roll3(dice = 1:6)` function will pass the value of dice as 1:6 and results will be computed.
- `roll3(dice = 1:12)` function will pass the value of dice as 1:6 and results will be computed but since we are computing the rolls of dice and a die has only 6 faces, for values 7:12 it returns GIGO.
- `roll3(dice = 1:24)` function will pass the value of dice as 1:6 and results will be computed but since we are computing the rolls of dice and a die has only 6 faces, for values 7:24 it returns GIGO.
- we should be wary when we are passing arguments without knowing context.

```
# Function in R: Continued...

# function which prints the chunk of given character vector best_practice
best_practice <- c('Let', 'the', 'computer', 'do', 'the', 'work')
print_words <- function(sentence) {
  print(sentence[1])
  print(sentence[2])
  print(sentence[3])
  print(sentence[4])
  print(sentence[5])
  print(sentence[6])
}

print_words(best_practice)
```

Code Sample 10

```
## [1] "Let"
## [1] "the"
## [1] "computer"
## [1] "do"
## [1] "the"
## [1] "work"
```

```
print_words(best_practice[-6])
```

```
## [1] "Let"
## [1] "the"
## [1] "computer"
## [1] "do"
## [1] "the"
## [1] NA
```

```
best_practice[-6]
```

```
## [1] "Let"      "the"      "computer" "do"       "the"
```

Interpretation:

- best_practice is variable vector of characters.
- print_words is function which takes sentence as argument and it prints the value at certain index of sentence argument.
- `print_words(best_practice)` calls the function and prints all of the words present in the best_practice using index.
- print_words(best_practice[-6]) calls the function and prints all of the words present in the best_practice using index except the last one which gives NA since `[-6]` removes the last value of best_practice
- `best_practice[-6]` removes the last from the vector.
- this is a very bad use of function as it consumes more memory because of multiple function calls also what if the size of vector or list changes, in that case after 6th index this function won't work properly.

```
# for loops in R
# for (variable in collection) {
#   do things with variable
# }
print_words <- function(sentence) {
  for (word in sentence) {
    print(word)
  }
}

print_words(best_practice)
```

Code Sample 11

```
## [1] "Let"
## [1] "the"
## [1] "computer"
## [1] "do"
## [1] "the"
## [1] "work"
```

```
print_words(best_practice[-6])
```

```
## [1] "Let"
## [1] "the"
## [1] "computer"
## [1] "do"
## [1] "the"
```

Interpretation:

- The previous problem can be solved using loops.
- print_words is function which prints the word in a sentence which is a argument for the print_words function.
- After calling `print_words(best_practice)` it prints each word in sentence.
- `print_words(best_practice[-6])` prints each word in sentence except the value at `[6]` index as [-6] removes the value at that index
- However, in R we should not used loops whenever possible as it is not efficient way to iterate.
- Using `apply`, `sapply`, `vapply` and `lapply` is an very good alternative of for loops in R.

```
# Conditionals in R

if (condition) {
  #code executed when condition is TRUE
} else {
  #code executed when condition is FALSE
}
```

Code Sample 12 ***Interpretation***:

- Simple if else conditional syntax in R which computer

```
# Conditionals in R
if (y1 < 20) {
  x <- "Too low"
} else {
  x <- "Too high"
}
```

Code Sample 13 ***Interpretation***:

- Simple if else condition syntax when y $<$ 20 the x is 'Too low' else x is 'Too High'.
- However, this code will throw error as y is not defined.

```r
# if else in R
check.y <- function(y) {
  if (y < 20) {
    print("Too Low")
  } else {
    print("Too High")
  }
}

check.y(10)
```

Code Sample 14

```
## [1] "Too Low"
```

```r
check.y(30)
```

```
## [1] "Too High"
```

Interpretation:

- Simple function `check.y` which checks values of y.
- If `y < 20` prints 'Too Low' else prints 'Too high'.
- check.y(10) prints 'Too Low' as it satisfies the if condition.
- check.y(10) prints 'Too High' as it satisfies the else condition.

```r
# binary variables using `ifelse`
y <- 1:40
ifelse(y < 20, 'Too low', 'Too high')
```

Code Sample 15

```
##  [1] "Too low"  "Too low"  "Too low"  "Too low"  "Too low"  "Too low"
##  [7] "Too low"  "Too low"  "Too low"  "Too low"  "Too low"  "Too low"
## [13] "Too low"  "Too low"  "Too low"  "Too low"  "Too low"  "Too low"
## [19] "Too low"  "Too high" "Too high" "Too high" "Too high" "Too high"
## [25] "Too high" "Too high" "Too high" "Too high" "Too high" "Too high"
## [31] "Too high" "Too high" "Too high" "Too high" "Too high" "Too high"
## [37] "Too high" "Too high" "Too high" "Too high"
```

```r
# logical
ifelse(y < 20, TRUE, FALSE)
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE
```

```r
# binary
y <- 1:40
ifelse(y < 20, 1, 0)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [39] 0 0
```

Interpretation:

- y is vector of sequence 1:40
- `ifelse` is very useful for binary values, `ifelse(y<20, 'Too low', 'Too high')` gives 'Too Low' for all values of y less than 20 and gives 'Too High' for all values of y greater than 20.
- `ifelse(y<20, TRUE, FALSE)` gives TRUE for all values of y less than 20 and gives FALSE for all values of y greater than 20.
- `ifelse(y<20, 1, 0)` gives 1 for all values of y less than 20 and gives 0 for all values of y greater than 20.
- very useful when creating categorical variable given some condition.

```r
# multiple if else conditions
if (this) {
  # do that
} else if (that) {
  # do something else
} else if (that) {
  # do something else
} else{
  # remaining
}
```

Code Sample 16 ***Interpretation***:

- simple syntax for multiple if else condition in R

```r
check.x <- function(x=1:99){
  if (x<20){
    print("Less than 20")
  }
  else {
    if (x < 40){
      print("20-39")
    }
    else {
```

```
      print("40-99")
    }
  }
}

check.x(15)
```

Code Sample 17

```
## [1] "Less than 20"
```

```
check.x(30)
```

```
## [1] "20-39"
```

```
check.x(45)
```

```
## [1] "40-99"
```

Interpretation:

- check.x is function that checks the value of x of given condition.
- `check.x(15)` prints '*Less than 20*' as it satisfies x $<$ 20.
- `check.x(30)` prints '*20-39*' as it satisfies x $<$ 40 in else block.
- `check.x(45)` prints '*40-99*' as it satisfies else condition in root else block which is x between 40 to 99.

```
x <- 1:99
x1 <- ifelse(x < 20, 1, 0)
x2.1 <- ifelse(x < 20, '<20', '20+')

x
```

Code Sample 18

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
## [51] 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
## [76] 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
```

```
x1
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [39] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [77] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
x2.1
```

```
##  [1] "<20" "<20" "<20" "<20" "<20" "<20" "<20" "<20" "<20" "<20" "<20" "<20"
## [13] "<20" "<20" "<20" "<20" "<20" "<20" "<20" "20+" "20+" "20+" "20+" "20+"
## [25] "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+"
## [37] "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+"
## [49] "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+"
## [61] "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+"
## [73] "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+"
## [85] "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+" "20+"
## [97] "20+" "20+" "20+"
```

Interpretation:

- `x <- 1:99` creates a sequence x containing integers from 1 to 99.
- `x1 <- ifelse(x<20, 1, 0)`, here we use the ifelse function. If the value of x is less than 20, x1 is assigned the value 1; otherwise, it's assigned 0.
- `x2.1 <- ifelse(x<20, '<20', '20+')`, here also we use the ifelse function. x2.1 is assigned the string '<20' if x is less than 20; otherwise, it's assigned '20+'.

```
x3 <- ifelse(x < 20, 1, ifelse(x < 40, 2, 3))
x3
```

Code Sample 19

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
## [39] 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
## [77] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

```
# represents the frequency distribution of categorical  data.
# It essentially counts the occurrences of unique values within a dataset and presents the result in a
table(x3)
```

```
## x3
##  1  2  3
## 19 20 60
```

Interpretation:

- `x3 <- ifelse(x<20, 1, ifelse(x<40, 2, 3))` In this line, we use nested `ifelse` statements to create a new vector x3
  - If the value of x is less than 20, x3 is assigned the value 1.
  - Otherwise, if the value of x is less than 40, x3 is assigned the value 2.
  - If neither condition is met (i.e., x is greater than or equal to 40), x3 is assigned the value 3.
- `table(x3)` generates a frequency table for the values in x3, showing how many times each value appears.

```r
iris <- within(iris, {
  Petal.cat <- NA
  Petal.cat[Petal.Length < 1.6] <- "Small"
  Petal.cat[Petal.Length >= 1.6 &
              Petal.Length < 5.1] <- "Medium"
  Petal.cat[Petal.Length >= 5.1] <- "Large"
})

iris$Petal.cat
```

Code Sample 20

```
##   [1] "Small"  "Small"  "Small"  "Small"  "Small"  "Medium" "Small"  "Small"
##   [9] "Small"  "Small"  "Small"  "Medium" "Small"  "Small"  "Small"  "Small"
##  [17] "Small"  "Small"  "Medium" "Small"  "Medium" "Small"  "Small"  "Medium"
##  [25] "Medium" "Medium" "Medium" "Small"  "Small"  "Medium" "Medium" "Small"
##  [33] "Small"  "Small"  "Small"  "Small"  "Small"  "Small"  "Small"  "Small"
##  [41] "Small"  "Small"  "Small"  "Medium" "Medium" "Small"  "Medium" "Small"
##  [49] "Small"  "Small"  "Medium" "Medium" "Medium" "Medium" "Medium" "Medium"
##  [57] "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium"
##  [65] "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium"
##  [73] "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium"
##  [81] "Medium" "Medium" "Medium" "Large"  "Medium" "Medium" "Medium" "Medium"
##  [89] "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium" "Medium"
##  [97] "Medium" "Medium" "Medium" "Medium" "Large"  "Large"  "Large"  "Large"
## [105] "Large"  "Large"  "Medium" "Large"  "Large"  "Large"  "Large"  "Large"
## [113] "Large"  "Medium" "Large"  "Large"  "Large"  "Large"  "Large"  "Medium"
## [121] "Large"  "Medium" "Large"  "Medium" "Large"  "Large"  "Medium" "Medium"
## [129] "Large"  "Large"  "Large"  "Large"  "Large"  "Large"  "Large"  "Large"
## [137] "Large"  "Large"  "Medium" "Large"  "Large"  "Large"  "Large"  "Large"
## [145] "Large"  "Large"  "Medium" "Large"  "Large"  "Large"
```

```r
table(iris$Petal.cat)
```

```
##
##  Large Medium  Small
##     42     71     37
```

Interpretation:

- `iris <- within(iris, { ... })` modifies the iris dataset by adding a new variable called Petal.cat. The within function allows us to create or modify variables within a data frame in this case, the iris data frame.
- `Petal.cat[Petal.Length < 1.6] <- "Small"`if the Petal.Length is less than 1.6, assign the category "Small" to Petal.cat.
- `Petal.cat[Petal.Length >= 1.6 & Petal.Length < 5.1] <- "Medium"` if the Petal.Length is between 1.6 (inclusive) and 5.1 (exclusive), assign the category "Medium" to Petal.cat.
- `Petal.cat[Petal.Length >= 5.1] <- "Large" if the Petal.Length` is greater than or equal to 5.1, assign the category "Large" to Petal.cat.
- `iris$Petal.cat` displays the values of the newly created Petal.cat variable for each observation(Small, Medium, Large) in the iris dataset.

- ‘table(iris$Petal.cat)1 generates a frequency table showing how many times each category appears in the Petal.cat variable.

```
if (temp <= 0) {
  "freezing"
}
else if (temp <= 10) {
  "cold"
}
else if (temp <= 20) {
  "cool"
}
else if (temp <= 30) {
  "warm"
}
else {
  "hot"
}
```

Code Sample 21 ***Interpretation***:

- Throws an error as temp object is not defined.

```
# Multiple Conditions: If, else if, else if, else if
x <- function(temp) {
  if (temp <= 0) {
    "freezing"
  }
  else if (temp <= 10) {
    "cold"
  }
  else if (temp <= 20) {
    "cool"
  }
  else if (temp <= 30) {
    "warm"
  }
  else {
    "hot"
  }
}
```

Code Sample 22 ***Interpretation***:

- as temp was not defined. I created a function x which takes temp as input.
- Inside the function, there are several conditions for temp statements (using if and else if):
  - if the value of temp is less than or equal to 0, the result is "freezing."
  - if the value of temp is between 0 (exclusive) and 10 (inclusive), the result is "cold."
  - if the value of temp is between 10 (exclusive) and 20 (inclusive), the result is "cool."
  - if the value of temp is between 20 (exclusive) and 30 (inclusive), the result is "warm."
  - if none of the above conditions are met (i.e., temp is greater than 30), the result is "hot."