UNIT-4

Object-Oriented Programming:

Object-Oriented Programming (OOP) along with R's objects, different classes like S3 and S4, along with its construction, creating its generic function with examples and many more.

S3 Classes:

- In R programming, S3 (Simple Scalar) classes are a simple and flexible system for defining and working with classes of objects.
- S3 classes allow you to associate attributes with an object, and to define methods that can operate on objects of that class.
- To create an S3 class, you can define a constructor function that creates an object with the desired attributes.
- The class of an object can be set using the class() function, and methods can be defined using generic functions that are named based on the class of the object they operate on.
- For example, to create an S3 class called myclass with a constructor function myclass() that sets an attribute x, you could use the following code:

```
myclass <- function(x) {
obj <- list(x = x)
class(obj) <- "myclass"
obj
}
```

- You can then define methods for the myclass class using generic functions like print() and summary(). For example:

```
print.myclass <- function(x, ...) {
  cat("My class object with attribute x =", x$x, "\n")
}

summary.myclass <- function(object, ...) {
  sum
mary(object$x)
}
```

- Overall, S3 classes are a useful and lightweight way to define custom classes in R, and can be used to simplify your code and make it more modular and reusable.

S4 Classes:

- S4 Class is stricter, conventional, and closely related to Object-Oriented concepts. The classes are represented by the formal class definitions of S4.
- More specifically, S4 has setter and getter functions for methods and generics. As compared to the S3 class, S4 can be able to facilitate multiple dispatches.
- In R programming, S4 (System 4) classes are another system for defining and working with classes of objects. S4 classes are more structured and formal than S3 classes, and are generally used for more complex object-oriented programming tasks.
- To create an S4 class, you define a formal definition of the class, including the slots (attributes) that the class will contain, and any methods (functions) that operate on objects of that class.
- Here's an example of an S4 class definition:

```
setClass(
  "Person",
  slots = list(
    name = "character",
    age = "numeric"
  ),
  prototype = list(
    name = "unknown",
    age = NA_real_
  )
)
```

- In this example, we define an S4 class called Person, with two slots: name and age. We also provide a prototype that specifies default values for the slots.
- Overall, S4 classes are a more formal and structured way of defining classes in R, and are often used for more complex object-oriented programming tasks. While they require more setup than S3 classes, they can be more powerful and flexible when used correctly.

S3 Versus S4 :

- In R programming, there are two main systems for defining classes of objects: S3 (Simple Scalar) classes and S4 (System 4) classes.
- Both S3 and S4 classes are used for object-oriented programming in R, but they have some important differences.
- S3 classes are simpler and more flexible than S4 classes. S3 classes are defined using generic functions that operate on objects of a given class. The class of an object is defined using the class() function.

- S3 classes do not have a formal class definition, but rather consist of a named vector or list with additional attributes.
- Methods for S3 classes are defined using functions with the naming convention generic.class. For example, the print() method for S3 class myclass would be defined as print.myclass.
- S4 classes, on the other hand, are more structured and formal than S3 classes. S4 classes have a formal definition that includes slots (attributes) and methods (functions) that operate on objects of that class.
- Slots are defined using the setClass() function, and methods are defined using the setMethod() function. S4 classes provide stronger typing and validation of objects.

Here are some key differences between S3 and S4 classes:

1. **Formal definition:** S4 classes have a formal definition that includes slots and methods, while S3 classes do not.
2. **Typing:** S4 classes provide stronger typing and validation of objects, while S3 classes do not. S4 classes allow you to specify the type of each slot, while S3 classes do not.
3. **Inheritance:** S4 classes support inheritance, while S3 classes do not. Inheritance allows you to define a new class that is based on an existing class, inheriting its slots and methods.
4. **Flexibility:** S3 classes are more flexible than S4 classes. S3 classes can be used to define classes with a wide variety of structures, while S4 classes require a more structured approach.

- Overall, the choice between S3 and S4 classes depends on the complexity of the programming task at hand.
- S3 classes are simpler and more flexible, while S4 classes are more structured and provide stronger typing and validation.
- S4 classes may be more appropriate for larger and more complex projects, while S3 classes may be more appropriate for smaller and simpler projects.

Managing Your Objects:

As a typical R session progresses, you tend to accumulate a large number of objects. Various tools are available to manage them. Here, we'll look at the following:

• The ls() function

• The rm() function

• The save() function

• Several functions that tell you more about the structure of an object, such as class() and mode()

• The exists() function

Listing Your Objects with the ls() Function:

- ✓ The ls() command will list all of your current objects.
- ✓ A useful named argument for this function is pattern, which enables wildcards.
- ✓ Here, you tell ls() to list only the objects whose names include a specified pattern. The following is an example.

> ls() [1] "acc" "acc05" "binomci" "cmeans" "divorg" "dv" [7] "fit" "g" "genxc" "genxnt" "j" "lo" [13] "out1" "out1.100" "out1.25" "out1.50" "out1.75" "out2" [19] "out2.100" "out2.25" "out2.50" "out2.75" "par.set" "prpdf" [25] "ratbootci" "simonn" "vecprod" "x" "zout" "zout.100" [31] "zout.125" "zout3" "zout5" "zout.50" "zout.75" > ls(pattern="ut") [1] "out1" "out1.100" "out1.25" "out1.50" "out1.75" "out2" [7] "out2.100" "out2.25" "out2.50" "out2.75" "zout" "zout.100" [13] "zout.125" "zout3" "zout5" "zout.50" "zout.75"

Removing Specific Objects with the rm() Function:

- ✓ To remove objects you no longer need, use rm(). Here's an example:

  > rm(a,b,x,y,z,uuu)

- ✓ This code removes the six specified objects (a, b, and so on). One of the named arguments of rm() is list, which makes it easier to remove multiple objects.
- ✓ This code assigns all of our objects to list, thus removing everything:

  > rm(list = ls())

- ✓ Using ls()'s pattern argument, this tool becomes even more powerful.

Saving a Collection of Objects with the save() Function:

Calling save() on a collection of objects will write them to disk for later retrieval by load(). Here's a quick example:

```
> z <- rnorm(100000)
> hz <- hist(z) > save(hz,"hzfile")
> ls() [1] "hz" "z"
> rm(hz) > ls() [1] "z"
> load("hzfile")
> ls() [1] "hz" "z"
 > plot(hz) # graph window pops up
```

The exists() Function:

The function exists() returns TRUE or FALSE, depending on whether the argument exists. Be sure to put the argument in quotation marks. For example, the following code shows that the acc object exists:

```
> exists("acc") [1] TRUE
```

Input/Output:

Accessing the Keyboard and Monitor:

R provides several functions for accesssing the keyboard and monitor. Here, we'll look at the scan(), readline(), print(), and cat() functions.

Using the scan() Function:

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list.

Suppose we have files named z1.txt, z2.txt, z3.txt, and z4.txt.

The z1.txt file contains the following:

```
123 4 5 6
```

The z2.txt file contents are as follows:

```
123 4.2 5 6
```

The z3.txt file contains this:

abc de f g

And finally, the z4.txt file has these contents:

abc 123 6 y

Let's see what we can do with these files using the scan() function.

> scan("z1.txt") Read 4 items [1] 123 4 5 6

> scan("z2.txt") Read 4 items [1] 123.0 4.2 5.0 6.0

> scan("z3.txt")

Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, : scan() expected 'a real', got 'abc'

> scan("z3.txt",what="")

Read 4 items [1] "abc" "de" "f" "g"

> scan("z4.txt",what="") Read 4 items [1] "abc" "123" "6" "y"

Using the readline() Function:

If you want to read in a single line from the keyboard, readline() is very handy.

> w <- readline()

abc de f

> w

[1] "abc de f"

Printing to the Screen:

At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression.

This won't work if you need to print from within the body of a function. In that case, you can use the print() function, like this:

> x <- 1:3

> print(x^2)

[1] 1 4 9

Reading and Writing Files:

Reading Files:

read.table() or read.csv():

These functions are used to read tabular data in R. Both functions work in a similar way, but read.csv() assumes that the file is comma-separated.

```
# Read a CSV file
data <- read.csv("data.csv")
```

readLines():

This function is used to read text files line by line.

```
# Read a text file
text <- readLines("file.txt")
```

scan():

This function is used to read files with specific format, like fixed-width files or files with different delimiters.

```
# Read a fixed-width file
data <- scan("file.txt", what=list("", "", 0), sep="")
```

readRDS():

This function is used to read R objects that have been saved using the saveRDS() function.

```
# Read an RDS file
data <- readRDS("data.rds")
```

Writing Files:

write.table() or write.csv():

These functions are used to write tabular data to a file. Both functions work in a similar way, but write.csv() writes comma-separated files by default.

```
# Write a CSV file
write.csv(data, "data.csv")
```

writeLines():

This function is used to write text files line by line.

```
# Write a text file
writeLines(text, "file.txt")
```

saveRDS():

This function is used to save R objects in a binary format that can be read later using the readRDS() function.

```
# Save an RDS file
saveRDS(data, "data.rds")
```

Accessing the Internet:

R's socket facilities give the programmer access to the Internet's TCP/IP protocol. For readers who are not familiar with this protocol, we begin with an overview of TCP/IP.

```
1 # set up socket connections with clients
2 #
3 cons <<- vector(mode="list",length=ncon) # list of connections
4 # prevent connection from dying during debug or long compute spell
5 options("timeout"=10000)
6 for (i in 1:ncon) {
7 cons[[i]] <<-
8 socketConnection(port=port,server=TRUE,blocking=TRUE,open="a+b")
9 # wait to hear from client i
10 checkin <- unserialize(cons[[i]])
11 }
12 # send ACKs
13 for (i in 1:ncon) {
14 # send the client its ID number, and the group size
15 serialize(c(i,ncon),cons[[i]]) 16 }
```

String Manipulation:

An Overview of String- Manipulation Functions:

In R programming, there are many built-in functions for manipulating strings. Here is an overview of some commonly used functions:

paste() and paste0():

These functions concatenate strings together. paste() adds a separator between strings while paste0() does not.

```
# concatenate strings with separator
paste("hello", "world", sep=" ")


# concatenate strings without separator
paste0("hello", "world")
```

substr():

This function extracts a substring from a string based on the start and end positions.

```
# extract a substring
substr("hello world", start=2, stop=5)
```

nchar():

This function returns the number of characters in a string.

```
# get the number of characters in a string
nchar("hello world")
```

toupper() and tolower():

These functions convert a string to uppercase or lowercase.

```
# convert a string to uppercase
toupper("hello world")


# convert a string to lowercase
tolower("HELLO WORLD")
```

Regular Expressions:

- ✓ Regular expressions (regex or regexp) are a powerful tool for working with text in R programming.
- ✓ Regular expressions allow you to search for patterns in strings and manipulate text based on those patterns.
- ✓ Here is an overview of how to use regular expressions in R:

grep() and grepl():

These functions search for a pattern in a vector of strings. grep() returns the indices of matching elements, while grepl() returns a logical vector indicating which elements match the pattern.

```
# Find all strings containing "hello"
x <- c("hello world", "foo bar", "hello there")
grep("hello", x)
grepl("hello", x)
```

gsub() and sub():

These functions perform find and replace operations on a string. gsub() replaces all instances of a pattern while sub() replaces only the first instance.

```
# Replace all instances of "o" with "a"
gsub("o", "a", "hello world")
# Replace the first instance of "o" with "a"
sub("o", "a", "hello world")
```

Regular expression syntax:

Regular expressions use special characters to represent patterns. Here are some commonly used characters:

- "." (dot): Matches any single character except a newline.
- "^" (caret): Matches the start of a string.
- "$" (dollar sign): Matches the end of a string.
- "*" (asterisk): Matches zero or more occurrences of the preceding character or group.
- "+" (plus): Matches one or more occurrences of the preceding character or group.

- "?" (question mark): Matches zero or one occurrences of the preceding character or group.
- "|" (vertical bar): Matches either the expression before or after the vertical bar.

```
# Find all strings starting with "h"
grep("^h", x)
# Find all strings ending with "rld"
grep("rld$", x)
# Find all strings containing "l" followed by "o"
grep("l.o", x)
```

Regular expression quantifiers:

Quantifiers are used to specify the number of occurrences of a pattern. Here are some commonly used quantifiers:

- {n}: Matches exactly n occurrences of the preceding character or group.
- {n,}: Matches n or more occurrences of the preceding character or group.
- {,m}: Matches up to m occurrences of the preceding character or group.
- {n,m}: Matches between n and m occurrences of the preceding character or group.

```
# Find all strings containing at least two "l"s
grep("l{2,}", x)
```