Loading and saving of data
S3, S4 and Reference Class

Take Input From User:

When we are working with R in an interactive session, we can use readline() function to take input from the user.

```
>name <- readline(prompt="Enter your name: ")
>Enter your name: MMDU

>print(name)
>[1] “MMDU“
```

Multiple inputs from user using R:

```
{name <- readline(prompt="E nter your name: ");
age <- readline(prompt="Enter your age: ")}

#OR

{name <- readline("Enter your name: ");
age <- readline("Enter your age: ")}
```

Enter your name: ram
Enter your age: 23

> print(name)
[1] "ram"
> print(age)
[1] "23“

Getting and Setting the Working Directory

You can check which directory the R workspace is pointing to using the **getwd()** function. You can also set a new working directory using **setwd()**function.

```
# Get and print current working directory.

> print(getwd())
[1] "C:/Users/HP World/Documents"

# Set current working directory.
setwd("/web/com“)

# Get and print current working directory.
print(getwd())
```

Reading and Writing from CSV File

Reading a CSV File

```
data <- read.csv(“filename")
print(data)
```

Analyzing the CSV File

```
print(is.data.frame(data))      #Returns True or False
print(ncol(data))
print(nrow(data))
```

Writing into a CSV File

```
write.csv(object,"outputfilename.csv")
```

Save and load data

```
x <- c(1:10)                    # create a numeric vector
y <- c(11:20)                   # create a numeric vector
z <- c(21:30)                   # create a numeric vector
m <- cbind(x, y, z)             # create a matrix
d <- as.data.frame(m)           # create a data frame

# create a text vector
t <- c("red", "blue", "red", "white", "blue", "white", "red","blue", "white", "white")
df <- cbind(d, t)               # add the text vector to the data frame
```

You can save the data frame **df** using this command:

save(df, file = "df.RData")

Load function reloads the data objects saved with save() function.

Load(“df.Rdata”);

```
>write.csv(df, file = "df.csv")

> z1<-read.csv("df.csv");
> z1
   X  x  y  z     t
1   1  1 11 21   red
2   2  2 12 22  blue
3   3  3 13 23   red
4   4  4 14 24 white
5   5  5 15 25  blue
6   6  6 16 26 white
7   7  7 17 27   red
8   8  8 18 28  blue
9   9  9 19 29 white
10 10 10 20 30 white
```

Classes and Methods

•Everything in R is an OBJECT.

• A class is the definition of an object.

• A method is a function that performs specific calculations on objects of a specific class. A generic function is used to determine the class of its arguments and select the appropriate method. A generic function is a function with a collection of methods.

• See ?Classes and ?Methods for more information.

Example - Classes and Methods

• For example, consider the ToothGrowth dataset that is included with R. This dataset looks at the effect of vitamin C on tooth growth in guinea pigs. (This is an example of S3 classes and methods.)

• ToothGrowth is an object, the class of ToothGrowth is a dataframe

• summary() is a generic function that calls the method summary.data.frame() to summarize the contents of ToothGrowth

```
class(ToothGrowth)
summary(ToothGrowth)
summary.data.frame(ToothGrowth)
```

S3 and S4 Classes

- There are two types of classes in R,
- **S3 Classes** - **old style, quick and dirty, informal**
- **S4 Classes - new style, rigorous and formal**

- The S3 approach is very simple to implement. Just add a class name to an object. There are no formal requirements about the contents of the object, we just expect the object to have the right information.

- An S4 class gives a rigorous definition of an object. Any valid object of an S4 class will meet all the requirements specified in the definition.

- Despite the shortcomings of S3 classes they are widely used and unlikely to disappear.

Why Classes and Methods?

•S4 classes reduce errors. When a function acts on a S4 class object it knows what type of information is in the object and that the information is valid.

•Methods simplify function calls and make computations more natural and clearer. Consider summary(), with methods we no longer need a separate function to summarize each type of object. Instead we can have one function that we call for summarizing a variety of object types.

•With S4 classes it is easier to organize and coordinate the work of multiple contributors for large complicated projects.

•For example, The bioconductor package used for microarray data makes extensive use of S4 classes and methods. (Standard Example)

S3 Class and Methods

•To create an S3 class all we need to do is set the class attribute of the object using class().

•To create an S3 method write a function with the name generic.class, where generic is a generic function name and class is the corresponding class for the method.

•Examples of generic functions are summary(), print() and plot().
See ?UseMethod for how to create new generic functions for S3 methods.

class(x)	Get or set the class attributes of x
unclass(x)	Remove class attributes of x
methods(generic.function)	All available S3 methods for a generic function
methods(class="class")	Get all S3 methods for a particular class
is(object)	Return object's class and all super-classes
is(object, class2)	Tests if object is from class2
str(object)	Display the internal structure of an object

Constructing a new S3 class

```
> laptop<- list(Company="Dell", Price=50000, Harddisk="500 GB")  # Create Instance of Class
> class(laptop)<-"Laptop“                    # Object of Class
> class(laptop)
[1] "Laptop"

> attributes(laptop)
$names
[1] "Company"  "Price"    "Harddisk"

$class
[1] "Laptop"
```

```
> laptop
$Company
[1] "Dell"

$Price
[1] 50000

$Harddisk
[1] "500 GB"

attr(,"class")
[1] "Laptop"
```

Checking the type of generic functions using ftype()

```
> methods(mean)
[1] mean.Date     mean.default  mean.difftime mean.POSIXct  mean.POSIXlt

> methods(plot)
 [1] plot.acf*          plot.data.frame*    plot.decomposed.ts* plot.default        plot.dendrogram*
plot.density*       plot.ecdf           plot.factor*        plot.formula*
[10] plot.function      plot.hclust*        plot.histogram*     plot.HoltWinters*   plot.isoreg*
plot.lm*            plot.medpolish*     plot.mlm*           plot.ppr*
[19] plot.prcomp*       plot.princomp*      plot.profile.nls*   plot.raster*        plot.spec*
plot.stepfun        plot.stl*           plot.table*         plot.ts
[28] plot.tskernel*     plot.TukeyHSD*

> methods(sum)
no methods found

> methods(max)
no methods found

> ftype(plot)
Error in ftype(plot) : could not find function "ftype“
```

[1] “S3” “generic”

S4 Class

S4 class are an improvement over the S3 class. They have a formally defined structure which helps in making object of the same class look more or less similar.
Class components are properly defined using the setClass() function and objects are created using the new() function.

```
# create a list with required components (S3 Class)
>s <- list(name = "John", age = 21, GPA = 3.5)

> # name the class appropriately
> class(s) <- "student“


# S4 Class
>setClass("student", slots=list(name="character", age="numeric", GPA="numeric"))
```

S4 objects are created using the new() function.

```
>s <- new("student",name="John", age=21, GPA=3.5)
>s An object of class "student"

Slot "name":
[1] "John"

Slot "age":
[1] 21

Slot "GPA":
[1] 3.5

>isS4(s)
[1] TRUE
```

```
># create an object using new()
># provide the class name and value for slots
```

```
# check for S4 is
```

Reference Class

Reference class were introduced later, compared to the other two. It is more similar to the object oriented programming we are used to seeing in other major programming languages.

Reference classes are basically S4 classed with an environment added to it. Instead of setClass() we use the setRefClass() function.

```
>setRefClass("student")
```

Member variables of a class, if defined, need to be included in the class definition. **Member variables of reference class are called fields** (analogous to slots in S4 classes).

```
> setRefClass("student", fields = list(name = "character", age = "numeric", GPA = "numeric"))
```

How to create a reference objects?

The function setRefClass() returns a generator function which is used to create objects of that class.

```
>student <- setRefClass("student", fields = list(name = "character", age = "numeric", GPA = "numeric"))

># now student() is our generator function which can be used to create new objects
 > s <- student(name = "John", age = 21, GPA = 3.5)

>s Reference class object of class "student"

Field "name":
[1] "John"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

How to access and modify fields?

Fields of the object can be accessed using the $ operator.

```
>s$name
[1] "John"

>s$age
[1] 21

>s$GPA
[1] 3.5
```

Similarly, it is modified by reassignment.

```
>s$name <- "Paul"

>s
Reference class object of class "student“

Field "name":
[1] "Paul"

Field "age":
[1] 21

Field "GPA":
[1] 3.5
```

Comparison between S3 vs S4 vs Reference Class

S3 Class	S4 Class	Referene Class
Lacks formal definition	Class defined using setClass()	Class defined using setRefClass()
Objects are created by setting the class attribute	Objects are created using new()	Objects are created using generator functions
Attributes are accessed using $	Attributes are accessed using @	Attributes are accessed using $
Methods belong to generic function	Methods belong to generic function	Methods belong to the class
Follows copy-on-modify semantics	Follows copy-on-modify semantics	Does not follow copy-on-modify semantics

```
>a <- list("x" = 1, "y" = 2)
>b <- a

# modify b
> b$y = 3

>a
 $x
[1] 1
$y
[1] 2

 > # only b is modified
>b
$x
 [1] 1
$y
[1] 3
```

But this is not the case with reference objects. Only a single copy exist and all variables reference to the same copy. Hence the name, reference.

```
>a <- student(name = "John", age = 21, GPA = 3.5)
>b <- a
```

```
>b$name <- "Paul“
```

```
> a

Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

```
>b

Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

This can cause some unwanted change in values and be the source of strange bugs. We need to keep this in mind while working with reference objects. To make a copy, we can use the copy() method made available to us.

```
>create reference object a and assign a's copy to b
>a <- student(name = "John", age = 21, GPA = 3.5)
>b <- a$copy()

>modify b
 > b$name <- "Paul"

>a remains unaffected
>a

Reference class object of class "student"
Field "name":
[1] "john"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```

```
>b

Reference class object of class "student"
Field "name":
[1] "Paul"
Field "age":
[1] 21
Field "GPA":
[1] 3.5
```