**1. Download any type of data (from the web or use datasets package) or create own set.**

**Solution:** R object S3 VS, S4

Code for S3

|  |
| --- |
| # Create a list object called "happy"  happy <- list(first = "one", second = "two", third = "third")  # Append the "monday" class to the existing class of "happy"  class(happy) <- append(class(happy), "monday")  # Print the "happy" list and its class attribute  happy  class(happy)  # Define a generic function called "GetFirst"  GetFirst <- function(x) {  UseMethod("GetFirst", x)  }  # Define a method for the "monday" class of objects  GetFirst.monday <- function(x) {  return(x$first)  }  # Call the "GetFirst" function on the "happy" object  GetFirst(happy) |

And the output is

|  |
| --- |
| # Create a list object called "happy"  > happy <- list(first = "one", second = "two", third = "third")  > # Append the "monday" class to the existing class of "happy"  > class(happy) <- append(class(happy), "monday")  > # Print the "happy" list and its class attribute  > happy  $first  [1] "one"  $second  [1] "two"  $third  [1] "third"  attr(,"class")  [1] "list" "monday"  > class(happy)  [1] "list" "monday"  > # Define a generic function called "GetFirst"  > GetFirst <- function(x) {  + UseMethod("GetFirst", x)  + }  > # Define a method for the "monday" class of objects  > GetFirst.monday <- function(x) {  + return(x$first)  + }  > # Call the "GetFirst" function on the "happy" object  > GetFirst(happy)  [1] "one" |

Code for S4:

|  |
| --- |
| # Define an S4 class called "Person"  setClass("Person",  slots = list(  name = "character",  age = "numeric"  )  )  # Define a generic function called "greeting"  setGeneric("greeting",  function(object) {  standardGeneric("greeting")  }  )  # Define a method for "greeting" with signature "Person"  setMethod("greeting",  signature = "Person",  function(object) {  paste("Hello, my name is", object@name, "and I am", object@age, "years old.")  }  )  # Create an object of class "Person"  my\_person <- new("Person", name = "John", age = 30)  # Call the "greeting" function on the "my\_person" object  greeting(my\_person) |

The output :

|  |
| --- |
| > # Define an S4 class called "Person"  > setClass("Person",  + slots = list(  + name = "character",  + age = "numeric"  + )  + )  > # Define a generic function called "greeting"  > setGeneric("greeting",  + function(object) {  + standardGeneric("greeting")  + }  + )  [1] "greeting"  > # Define a method for "greeting" with signature "Person"  > setMethod("greeting",  + signature = "Person",  + function(object) {  + paste("Hello, my name is", object@name, "and I am", object@age, "years old.")  + }  + )  > # Create an object of class "Person"  > my\_person <- new("Person", name = "John", age = 30)  > # Call the "greeting" function on the "my\_person" object  > greeting(my\_person)  [1] "Hello, my name is John and I am 30 years old." |

**2. second step, determine if generic function as discussed in this module can be assigned to your data set, and if not, why? (Example, here is list of data set in R)data("mtcars")  
head (mtcars, 6)  
list(mtcars, 6)**

**Solution:** The code provided loads the built-in mtcars dataset in R and displays the first six rows of the dataset. It does not define a generic function.

A generic function is a function that behaves differently based on the data type or class of the input object. For example, summary() is a generic function in R that provides a summary of the input object, and its behavior changes based on the class of the object.

To define a generic function, you need to use the UseMethod() function, which dispatches the function to a specific method based on the class of the input object. Here's an example:

|  |
| --- |
| my\_generic\_function <- function(x) {  UseMethod("my\_generic\_function")  }  my\_generic\_function.data.frame <- function(x) {  # do something with a data frame  }  my\_generic\_function.list <- function(x) {  # do something with a list  } |

In this example, my\_generic\_function() is a generic function that dispatches to specific methods based on the class of the input object. The my\_generic\_function.data.frame() method is called when the input object is a data frame, and the my\_generic\_function.list() method is called when the input object is a list.

However, in the code you provided, you are not defining a generic function. Instead, you are simply loading the mtcars dataset and displaying the first six rows.

**3. In the last step, explore if S3 and S4 can be assigned to your data set or the database "mtcars."**

**Solution:**  S3 and S4 are object-oriented programming systems in R that are used for defining and manipulating objects. These systems are typically used for creating complex data structures that can be used to store and manipulate data in a more organized and efficient manner.

In the case of a custom dataset or the "mtcars" database, S3 and S4 can certainly be used to define and manipulate objects. For example, you could define an S3 object that represents a single observation or row of data from your custom dataset, or an S4 object that represents the entire dataset itself.

To work with a custom dataset in R, you can first load the dataset into memory using a function like ‘read.csv ()’ or ‘read.table()’

. For example, if you have a CSV file called "mydata.csv" in your working directory, you can load it into R using the following code:

|  |
| --- |
| mydata <- read.csv("mydata.csv") |

This will load the "mydata.csv" file into a data frame object called "mydata" in R. You can then use S3 or S4 to define and manipulate objects based on this data frame.

For example, you could define an S3 object that represents a single observation or row of data from your custom dataset, like this:

|  |
| --- |
| setClass("mydata\_observation",  representation(  var1 = "numeric",  var2 = "numeric",  var3 = "numeric",  var4 = "numeric"  ))  mydata\_observation <- function(var1, var2, var3, var4) {  new("mydata\_observation",  var1 = var1,  var2 = var2,  var3 = var3,  var4 = var4)  } |

This code defines a new S3 class called "mydata\_observation" that represents a single observation or row of data from your custom dataset. The class has four attributes, each of which corresponds to a variable in your dataset. The mydata\_observation() function can then be used to create new objects of this class, like this:

|  |
| --- |
| obs <- mydata\_observation(var1 = 3.2, var2 = 1.5, var3 = 7.8, var4 = 2.6) |

This code creates a new object of the "mydata\_observation" class with the specified attribute values.

Similarly, you could define an S4 class that represents the entire custom dataset, like this:

|  |
| --- |
| setClass("mydata\_dataset",  representation(  data = "data.frame"  ))  mydata\_dataset <- function(data) {  new("mydata\_dataset",  data = data)  } |

This code defines a new S4 class called "mydata\_dataset" that represents the entire custom dataset. The class has a single attribute called "data" that contains the data frame object representing the dataset. The mydata\_dataset() function can then be used to create new objects of this class, like this:

|  |
| --- |
| data\_obj <- mydata\_dataset(data = mydata) |

This code creates a new object of the "mydata\_dataset" class with the entire dataset as its attribute.

In summary, S3 and S4 can certainly be used to define and manipulate objects based on a custom dataset or the "mtcars" database in R. The examples provided demonstrate how to define S3 and S4 classes and create objects based on them using the data from the dataset.

**Discuss the following questions:**

**1. How do you tell what OO system (S3 vs. S4) an object is associated with?**

**Solution:** In R, we can check whether an object was created using the S3 or S4 object system by using the ‘isS4()’ function. This function returns a logical value of ‘TRUE’ if the object was created using S4, and ‘FALSE’ otherwise.

For example, to check whether an object ‘my-object’ was created using S4, you can use the following code:

Ex Code : isS4(my\_object)

If the above code returns ‘TRUE’, then ‘my\_object’ was created using S4; if it returns ‘FALSE’, then ‘ my object’ was created using S3.

Alternatively, you can also check if an object was created using S3 by examining its class attributes using the ‘class()’ function. S4 objects typically have class attributes that begin with the string "Class". In contrast, S3 objects have a character vector of class names. Therefore, if an object has a class attribute that begins with "Class", it is likely an S4 object.

For example, to check the class attributes of an object ‘my\_object’, you can use the following code:

Ex Code : class(my\_object)

If the class attribute begins with "Class", then’ my\_object’ is likely an S4 object. If the class attribute is a character vector, then ‘my\_object’ is likely an S3 object.

**2. How do you determine the base type (like integer or list) of an object?**

In R, you can determine the base type of an object using the ‘class()’ function or the ‘typeof()’ function.

The ‘class()’ function returns the name of the object's class or classes, in the form of a character vector. For example, if you have an object named ‘x’, you can determine its class by calling ‘class(x)’.

The ‘typeof()’ function returns the basic type of the object as a string, such as "integer", "numeric", "character", "logical", "complex", "raw", "list", or "function". For example, if you have an object named ‘y’, you can determine its type by calling ‘typeof(y)’.

Here's an example that demonstrates the use of these functions:

# create some objects of different types

x <- 1:5

y <- "hello"

z <- list(a = 1, b = 2, c = 3)

# determine the types of these objects

typeof(x) # "integer"

typeof(y) # "character"

typeof(z) # "list"

# determine the classes of these objects

class(x) # "integer"

class(y) # "character"

class(z) # "list"

Note that the ‘typeof()’ function returns a basic type, while the ‘class()’ function can return more specific information about the object's class or classes.

**3. What is a generic function?**

In R, a generic function is a function that defines a common interface or behavior for a set of related functions, which can operate on different classes or types of objects. The generic function provides a consistent way to call and use these functions, regardless of the specific class or type of object being operated on.

For example, the "+" operator in R is a generic function that can be used to add numeric values, concatenate character strings, and combine other types of objects. The behavior of the "+" operator is determined by the class or type of the objects being operated on.

To create a new generic function in R, you can use the "UseMethod" function. Here's an example of a generic function "my\_func" that can operate on different types of objects:

my\_func <- function(x) {

UseMethod("my\_func")

}

my\_func.default <- function(x) {

message("Default behavior")

}

my\_func.numeric <- function(x) {

message("Numeric behavior")

}

my\_func.character <- function(x) {

message("Character behavior")

}

In this example, the "my\_func" function is a generic function that can operate on different classes of objects, including numeric and character. When "my\_func" is called with a numeric object, the "my\_func.numeric" function is called, and when it is called with a character object, the "my\_func.character" function is called. If it is called with any other type of object, the "my\_func.default" function is called, which provides a default behavior.

**4. What are the main differences between S3 and S4 ?**

In R, S3 and S4 refer to different object-oriented programming systems that are used to define and manipulate objects in R. They are not directly related to the Amazon Web Services (AWS) object storage service Amazon S3 (Simple Storage Service) mentioned in the previous question.

Here are some of the main differences between S3 and S4 in R:

|  |  |  |
| --- | --- | --- |
|  | S3 | S4 |
| Method dispatch | S3 dispatches methods based on the class of the first argument of the function | S4 dispatches methods based on the class of the object passed as the first argument. |
| Inheritance | S3 does not have a formal mechanism for inheritance. | S4 supports formal inheritance, where classes can inherit methods and properties from parent classes |
| Class definition | S3 classes can be defined more informally. | S4 requires that classes be defined with a formal class definition |
| Method definitions | S3 methods are defined more informally. | methods are defined with a S4 formal syntax that includes a signature specifying the arguments that the method expects |
| Error handling | S3 provides detailed error messages and error handling | S4 provides more detailed error messages and better error handling than S3. |
| Backward compatibility | some S3 code may not work with S4 without modification. | S4 is not fully backward compatible with S3 |

In summary, S4 is a more formal and complex object-oriented system in R that offers better support for inheritance, method dispatch, and error handling than S3. However, S3 is simpler to use and is more backward compatible with older R code. The choice between S3 and S4 often depends on the specific needs of the user and the requirements of the project.

**5.create two examples of S3 . Post each step you took to create and test the two functions useMethod() and setGeneric()**

**Solution;**

Sure, here are two examples of S3 and the steps to create and test generic and method functions using setGeneric() and useMethod() in R:

**Example 1:** S3 method for sorting a vector

Step 1: Define a generic function

In this example, we'll create a generic function called "sort\_vector" that takes a single argument of class "vector".

|  |
| --- |
| setGeneric("sort\_vector", function(x) {  standardGeneric("sort\_vector")  }) |

Step 2: Define the method functions

-Now we can create method functions for specific classes using the useMethod() function.

-Let's create a method function called "sort\_vector.numeric" for the "numeric" class.

|  |
| --- |
| sort\_vector.character <- function(x) {  return(sort(x))  } |

Step 3: Test the generic and method functions.

Now that we've defined our generic and method functions, we can create some objects of the relevant classes and test the functions.

|  |
| --- |
| # Create a numeric vector  x1 <- c(4, 2, 1, 3)  class(x1) <- "numeric"  # Create a character vector  x2 <- c("dog", "cat", "bird", "elephant")  class(x2) <- "character"  # Call the generic function with the objects as arguments  sort\_vector(x1) # 1 2 3 4  sort\_vector(x2) # "bird" "cat" "dog" "elephant" |

This will call the appropriate method function for each object and sort the vectors accordingly.

**Example 2**: S3 method for calculating the area of a shape

Step 1: Define a generic function

In this example, we'll create a generic function called "calculate\_area" that takes a single argument of class "shape".

|  |
| --- |
| setGeneric("calculate\_area", function(shape) {  standardGeneric("calculate\_area")  }) |

Step 2: Define the method functions

Now we can create method functions for specific classes using the useMethod() function.

Let's create a method function called "calculate\_area.rectangle" for the "rectangle" class.

|  |
| --- |
| calculate\_area.rectangle <- function(shape) {  area <- shape$length \* shape$width  return(area)  } |
|  |

Let's create another method function called "calculate\_area.circle" for the "circle" class.

|  |
| --- |
| calculate\_area.circle <- function(shape) {  area <- pi \* shape$radius^2  return(area)  } |

Step 3 : Test the generic and method functions

- - Now that we’ve defined our generic and method functions, We can create some objects of the relevant classes and test the functions.

|  |
| --- |
| # Create a rectangle object  rectangle <- list(length = 5, width = 3)  class(rectangle) <- "rectangle"  # Create a circle object  circle <- list(radius = 4)  class(circle) <- "circle"  # Call the generic function with the objects as arguments  calculate\_area(rectangle) # 15  calculate\_area(circle) # 50.26548 |

This will call the appropriate method function for each object and calculate the area of the shape.

5. In your GitHub, create two examples of S4. Each step I took to create and test the two functions  **useMethod()**and **setGeneric()**

here are two examples of S4 and the steps to create and test generic and method functions using setGeneric() and useMethod() in R:

**Example 1**: S4 class for a bank account

Step 1: Define the S4 class

In this example, we'll create an S4 class called "bank\_account" that has the following slots: "owner" (character), "balance" (numeric), and "account\_number" (integer).

|  |
| --- |
| setClass("bank\_account",  slots = c(  owner = "character",  balance = "numeric",  account\_number = "integer"  )  ) |

Step 2: Define the generic function

Now we can create a generic function called "deposit" that takes two arguments: an object of class "bank\_account" and the amount to be deposited.

|  |
| --- |
| setGeneric("deposit", function(object, amount) {  standardGeneric("deposit")  }) |

Step 3: Define the method function

-We can create a method function for the "bank\_account" class using the useMethod() function.

-Let's create a method function called "deposit.bank\_account" that adds the deposited amount to the account balance.

|  |
| --- |
| setMethod("deposit", "bank\_account", function(object, amount) {  object@balance <- object@balance + amount  return(object)  }) |

Step 4: Test the generic and method functions

Now that we've defined our S4 class and generic/method functions, we can create some objects of the relevant class and test the functions.

|  |
| --- |
| # Create a bank account object  account <- new("bank\_account",  owner = "John Smith",  balance = 1000,  account\_number = 123456  )  # Deposit $500 into the account  deposit(account, 500)  # Check the updated balance  account@balance # 1500 |

This will call the appropriate method function for the "bank\_account" object and update the balance accordingly.

**Example 2**: S4 class for a geometric shape

Step 1: Define the S4 class

In this example, we'll create an S4 class called "shape" that has the following slots: "name" (character), "area" (numeric), and "perimeter" (numeric).

|  |
| --- |
| setGeneric("display\_shape\_info", function(shape) {  standardGeneric("display\_shape\_info")  }) |

Step 3: Define the method function

We can create a method function for the "shape" class using the useMethod() function.

Let's create a method function called "display\_shape\_info.shape" that prints out the name, area, and perimeter of the shape.

|  |
| --- |
| setMethod("display\_shape\_info", "shape", function(shape) {  cat("Name:", shape@name, "\n")  cat("Area:", shape@area, "\n")  cat("Perimeter:", shape@perimeter, "\n")  }) |

Step 4: Test the generic and method functions

Now that we've defined our S4 class and generic/method functions, we can create some objects of the relevant class and test the functions.

|  |
| --- |
| # Create a rectangle object  rectangle <- new("shape",  name = "rectangle",  area = 15,  perimeter = 16  )  # Create a circle object  circle <- new |