R

Objects, Statistics, and Packages

Objects in R

- `R` supports three different types of objects, all declared and used in different ways
  - S3 objects
  - S4 objects
  - RC objects

S3 Objects

- S3 objects are the simplest and most common type of object in `R`
- Based of the design of objects in the third version of the `S` language
  - Came out in 1988
  - Switched from FORTRAN to C
- Methods don't belong to objects, uses a form of object-oriented programming known as generics

Creating an S3 Object

- Any existing object can be converted into an S3 object
  - Use the `structure` function and assign the results to a variable
  - Use the assignment version of the `class` function to give an existing variable a class attribute
- Both of these methods create a single instance at a time

In [ ]:

```
my_first_instance <- structure(1:5,class="specialVector")
print(my_first_instance)
print(str(my_first_instance))
```

In [ ]:

```
my_second_instance <- list(a_member = 2, another= "A String")
print(my_second_instance)

class(my_second_instance) <- "listClass"
print(str(my_second_instance))
```

S3 Constructor

- An S3 constructor is one that simply hides the call to `structure` or `class` inside of a function
- By convention, it should have the same name as the class, although this isn't strictly necessary

```
class_name <- function(parameters){
  structure(list(parameters),class="class_name")
  }
```

In [ ]:

```
vehicle <- function(n_wheels,color){
    structure(list(m_n_wheels = n_wheels, m_color = color ),
              class="vehicle")
}

myCar <- vehicle(4,'black')
print(class(myCar))
```

Inheritance

- The class attribute of an object cab actually be a vector
  - We can use this to simulate inheritance
  - In the previous examples, we are inheriting from the list class

```
child_class <- function(parameters)
{
  self <- parent_class(parameters)
  class(self) <- append("child_class",class(self))
  self
}
```

In [ ]:

```
car <- function(color){
    self <- vehicle(4,color)
    class(self) <- append("car",
                          class(self))
    self
}
my_new_car <- car('black')
print(class(my_new_car))
```

Methods

- `R` uses a style of OOP known as generics
  - An object is passed to a function, which then acts on the object
  - By writing multiple different "versions" of the same function, we can specify how the function should interact on a given object
- Most functions we have seen so far are actually generics, ie

```
t(df) # actually t.data.frame(df)
```

In [ ]:

```
mm <- as.data.frame(matrix(1:20,ncol=4))
print(t(mm))
print(t.data.frame(mm))
```

In [ ]:

```
print(t)

print(t.data.frame)
```

The Generic Function

- The top level function must be created and follows a very standard format.
  - The `UseMethod` function denotes that this function should actually dispatch to a more appropriate function, based on the object that was passed in
- The generic function for `t` might look like

```
t <- function(obj){
      UseMethod("t")
  }
```

User-Defined Generics

- Write a generic function with the name of the function you want
- For each class you want to define a different version of your function for, name it as `function_name.class_name`
  - The generic function will use the class attribute of the function passed to it to determine which to call
- A function named `function_name.default` can be defined to be run in the event no match is found

In [ ]:

```
print(my_new_car)
print.vehicle <- function(x)
{
    "My vehicle is " % % x[['m_color']] % % "in color and  has" % % x$m_n_wheels %
 % "wheels."
}
print(my_new_car)
#print.vehicle <- print.default
rm(print.vehicle)
print(my_new_car)
```

In [ ]:
```
makeNoise <- function(x){
    print(class(x))
    UseMethod("makeNoise")
}

makeNoise.vehicle <-function(x){
    "Generic Vehicle Noise"
}

makeNoise.car <- function(x){
    "BEEP BEEP"
}

makeNoise.default <- function(x){
    "You can't make a noise"
    }
```

In [ ]:

```
print(makeNoise(myCar))
print(makeNoise(my_new_car))
print(makeNoise("Random String"))
```

S3 Object Practice

- Make an S3 class that represents a book you are reading
  - The book has a title, a number of pages, and the page you are currently on, which is 1 to start with
  - Make a print method that prints a nice summary of the object
  - Make a read method, that takes in a number of pages, and increased the page you are currently on by that ammount

S4 Classes

- S4 is based on the object system from the 4th version of S, released in 1998
- Not as commonly found, but some more complex libraries do make uses of it
- Very similar to S3, but more formal
  - Classes must be initialized using the `new` function
  - The properties of the classes are part of the definition (called `slots` in `R`)
  - Inheritance is done through use of the `contains` keyword

Reference Classes

- Reference classes are the newest object system in `R`
  - Released around 2010
- Behave much more like traditional classes in other languages
  - Methods now belong to objects

Frequency

- Counting the frequency of an element in `R` is done using the various `table` functions
  - `table` returns a `table` object, which may be converted to a data frame for easier querying
- There is no limit to the number of variables in a cross-tabulation, although it is rare to see something beyond a 2 or 3 way frequency
  - To print higher dimension frequencies, pass table to `ftable`

Frequency of Qualitative Data

- Qualitative Data represents categories
  - No additional preprocessing needed with categorical data

In [ ]:

```
strings <- c("Yes","Yes","No","Maybe","OK","Yes")
print(table(strings))
```

In [ ]:

```
library(vcd)
head(Bundesliga)
```

In [ ]:

```
print(table(Bundesliga$HomeTeam))
```

In [ ]:

```
homeGames <- table(Bundesliga$HomeTeam)
print(head(homeGames[order(-homeGames)]))
```

In [ ]:

```
## How do we get the total number of games played?
away_games <- table(Bundesliga$AwayTeam)
all_games <- away_games + homeGames
print(head(all_games[order(-all_games)]))
```

In [ ]:

```
print(head(table(Bundesliga$HomeTeam,Bundesliga$AwayTeam)))
```

Frequency of Quantitative Data

- Quantitative Data requires preprocessing
  - The `table` function can only count things, it won't bin numbers for us
- The `cut` function converts numeric data into factors
  - In addition to the vector to cut, we can either pass the number of bins, or the bins themselves we want to use
  - The parameter `right` controls which side is open and which is closed

In [ ]:

```
print(max(Bundesliga$HomeGoals))
FactorGoals <- cut(Bundesliga$HomeGoals,3,right=FALSE)
print(table(FactorGoals))
```

In [ ]:

```
print(head(table(Bundesliga$HomeTeam,FactorGoals)))
```

In [ ]:

```
goalsByTeam <- as.data.frame(table(Bundesliga$HomeTeam,FactorGoals))
print(head(goalsByTeam))
```

In [ ]:

```
goalsByTeam <- as.data.frame.matrix(table(Bundesliga$HomeTeam,FactorGoals))
print(head(goalsByTeam))
```

In [ ]:

```
print(order(-goalsByTeam[3]))
print(head(goalsByTeam[order(-goalsByTeam[3]),]))
```

Descriptive Statistics

- Almost every basic statistical function is built-in in `R`
  - `mean`
  - `median`
  - `sd` - Standard Deviation
  - `max`
  - `min`

In [ ]:

```
print(paste("Our dataset includes the years from",
            min(Bundesliga$Year),"to",max(Bundesliga$Year)))
print(mean(Bundesliga$AwayGoals))
print(mean(Bundesliga$HomeGoals))
print(sd(Bundesliga$AwayGoals))
print(sd(Bundesliga$HomeGoals))
```

In [ ]:

```
sumAway <- summary(Bundesliga$AwayGoals)
print(class(sumAway))
print(sumAway)
print(summary(Bundesliga$HomeGoals))
```

Applying Over Axis

- When applying a descriptive function like mean to a matrix or array, the default option is to flatten it like a vector
- To apply is only over rows or only over columns, we need to use another function
  - For mean, there is the special functions `rowMeans` and `colMeans`
  - In general, we can use the `apply` function, which applies a function over an object across a given margin(sometimes called an axis)
    - In a matrix, 1 applies over the rows, and 2 applies over the columns

```
apply(OBJECT,AXIS,FUNCTION)
```

In [ ]:

```
library(psych)
#print(dim(iqitems))
#print(head(iqitems))
iqitems[is.na(iqitems)] <- 0
print(mean(as.matrix(iqitems)))
```

In [ ]:

```
print(apply(iqitems,2,mean))
```

Correlation

- There are many different kinds of correlation, three of the most common are
  - Pearson's r (most common)
  - Kendall's τ (Rank-based correlation)
  - Spearman ρ (Rank-based correlation)
- All are available in `R` using the `cor` method, and passing the corresponding string to the `method` parameter

In [ ]:

```
print(cor(Bundesliga$HomeGoals, Bundesliga$AwayGoals,method="spearman"))

## Not really useful because its comparing ranks, but this is how it is called
print(cor(Bundesliga$HomeGoals, Bundesliga$AwayGoals,method="kendall"))
```

PCA

- `R` also comes built in with numerous exploratory data techniques
- Principal Components Analysis (PCA) is a dimensional reduction technique that attempts to find the most important components
- The PCA function in R is named `prcomp`

In [ ]:

```
pca <- prcomp(iqitems)
print(pca$x)
```

K-Means

- Clustering is both a machine learning technique as well as a method of exploratory analysis
- The `kmeans` function produces k-clusters by using attributes of data
  - By default, it will use all attributes, if you don't want this, select a subset before passing it to K-means
- A `kmeans` object is returned

In [ ]:
```
clusters <- kmeans(iqitems,10)
print(clusters)
```

In [ ]:

```
print(str(clusters))
print(clusters$cluster)
```

In [ ]:

```
#clusters$cluster[clusters$cluster==2]
head(iqitems[names(clusters$cluster[clusters$cluster==2]),])
```

Linear Regression

- It is very common after some exploratory analysis to build a model in R
- Linear regression in `R` is performed using the `lm` function
- `lm` is the first function we are looking at that takes as an argument a formula

```
lm(formula, data = DATAFRAME)
```

Formulas in R

- A formula in R has the general form of

```
dependent_var ~ independent_vars
```

- Variable names are not quoted, and are expected to refer to columns in the data frame
- If you think there is no interaction between the independent variables, combine them using `+`
- If you think there is interaction, or just want to allow it as a possibility, combine them using `*`

In [ ]:
```
head(iris)
```

In [ ]:
```
model1 <- lm(Sepal.Length ~ Sepal.Width + Petal.Length, data = iris)
summary(model1)
```

In [ ]:

```
model2 <- lm(Sepal.Length ~ Sepal.Width * Petal.Length, data = iris)
summary(model2)
```

In [ ]:

```
model3 <- lm(Sepal.Length ~ Sepal.Width * Petal.Length * Species, data = iris)
summary(model3)
```

ANOVA

- In the social sciences, a very common anaylsis is to determine which variable is the most signifigant
  - The most common way to doing this is Analysis of Variance (ANOVA)
- ANOVA is actually a specialized version of a linear model, but we can call it explicitly by using the function `aov`
  - If you already have a linear model, you can print the ANOVA by using the function `anova`

In [ ]:

```
model4 <- aov(Sepal.Length ~ Sepal.Width * Petal.Length * Species,
              data = iris)
print(summary(model4))
```

In [ ]:

```
print(anova(model3))
```

Packages in R

- Like most scripting languages, `R` has a very robust package ecosystem
- To install a package in `R`, use the `install.packages` function, and pass the name of the function you want to install
- Once a package is installed, you can use it by calling

```
library(PACKAGE_NAME) #No QUOTES
```

Package Documentation

- Most major packages in R come with two forms of documentation
  - The manual, which contains the same information that can be accessed through the `?` operator
  - Vingettes, which is a more long form documentation, often written in the style of an academic paper
- Example
  - https://cran.r-project.org/web/packages/psych/psych.pdf
  - https://cran.r-project.org/web/packages/psych/vignettes/intro.pdf
  - https://cran.r-project.org/web/packages/psych/vignettes/overview.pdf

CRAN

- So where do the packages come from when we perform `install.packages`?
- By default the come from CRAN the Comprehensive R Archive Network
  - Most scripting languages have an equivalent, often named similarly (CTAN, CPAN)
- Other package repositories exist and can be used, but if you are using a popular package, it is probably published on CRAN

Finding Pacakges

- CRAN is great at hosting packages
  - Not great at helping you find packages
- Numerous third party websites exist to help you find a package to accomplish something
  - My personal favorite is https://crantastic.org/

TidyData

- There are many ways to represent data in a data frame, and due to the history of R, almost all of them are use
- Recently there has been a push to create commonsense conventions, known as having "Tidy Data"
- Hadley Wickham (Major player in R and the tidy data movement) defines tidy data as
  - Each variable is in a column.
  - Each observation is a row.
  - Each value is a cell.

TidyR

- To promote and enable this, the package TidyR was released
- It was spawned an entire family of packages, collectively known as the tidyverse
  - You can install just tidyR by using install.packages('tidyR')
  - The entire family can be installed with install.packages('tidyverse')
- It contains many functions meant to manipulate data into a tidy form

The Pipe Operator

- `TidyR` is commonly presented using the operator `%>%`, which comes from an earlier package, `magrittr`
  - It is very similar to the pipe in bash, passing the output of one function as the first argument to the next function
  - The following are eqiuvalent

```
apply(data,1,function)

data %>% apply(1,function)
```

Spreading

- The `spread` function converts from long data to wide data
- The syntax of the `spread` function is

  ```
  spread(data,key,value)
  ```

  - Key is the column you want to use to form your new columns
  - Value is the column you want to use to fill the cells

In [ ]:

```
library(DSR)
long <- table2
extra_wide_cases <- table4
combined <- table5
```

In [ ]:

```
library(tidyr)
print(as.data.frame(spread(long,?,?)))
```

Gathering

- Gathering is the opposite of spread
  - While it is uncommon to need this, it is possible someone made a data frame where not every column is a variable, and you need to collapse things a bit

```
gather(data, COLUMN_NAME1, COLUMN_NAME2, cols_to_gather)
```

In [ ]:

```
gathered_cases <- extra_wide_cases %>% gather("Year","Cases",2:3)
print(gathered_cases)
```

Separating and Uniting

- Separating and Uniting allows us to create multiple columns from one, or bring together columns that should never has been separated

```
separate(data,col_to_separate,new_columns)
  unite(data,col_to_add, from_columns)
```

In [ ]:

```
print(combined)
all_good <- combined %>% unite("year",?) %>% separate(?,?)
print(all_good)
```