Data Objects and Basics Data Visualization

STAT 220

Bastola
January 16 2022

Objects in R

- Anything created or imported into R is called an object
  - vectors, data frames, matrices, lists, functions, lm, ...

- We usually store objects in the workspace using the assignment **operator** '<-'

```
x <- c(8,2,1,3)
ls()
## [1] "x"
```

- The `=` operator also does assignment, but it is mainly used for argument specification inside a function.

```
y <- rnorm(3, mean = 10, sd = 2)
ls()
## [1] "x" "y"
```

- Please don't use `=` for assignment!!!

Data structures and types

- Every object is a **vector**
- `NULL` = empty object (vector of length 0)
- `typeof()`: tells us about storage of data
  - Logical: TRUE or FALSE
  - Integer and double
  - Character: string ("") of text
  - List

```
x <- c(8,2,1,3)
typeof(x)  # type of storage mode
## [1] "double"
typeof(c(8L,2L,1L,3L))  # adding L forces integer type
## [1] "integer"
x == 1
## [1] FALSE FALSE  TRUE FALSE
typeof(x == 1)
## [1] "logical"
```

Data structures and types

- Every object is a **vector**
- `NULL` = empty object (vector of length 0)
- `typeof()`: tells us about storage of data
  - Logical: TRUE or FALSE
  - Integer and double
  - Character: string ("") of text
  - List

`class` further describes the object

```
x <- c(8,2,1,3)
typeof(x)  # type of storage mode
## [1] "double"
typeof(c(8L,2L,1L,3L))  # adding L forces integer type
## [1] "integer"
x == 1
## [1] FALSE FALSE  TRUE FALSE
typeof(x == 1)
## [1] "logical"
```

```
class(x)   # object class is numeric
## [1] "numeric"
```

Object Oriented Programming

- In R, commands care about object class and type
    - Ex: the default `plot` command wants a vector of data or a formula to form a scatterplot

```
plot(y ~ x, data= mydata) # makes scatterplot if x and y numeric
```

- but if you give `plot` a `lm` regression object it will produce a set of diagnostic plots for that regression model.

```
my_lm <- lm(y ~ x, data= mydata)  # make a linear model
plot(my_lm)  # makes multiple diagnostic plots
```

- In your **Console** window, type `?plot` then hit **tab**.

> ?plot

```
- see plot, plot.acf, ...
```

Atomic Vectors and lists

- R uses two types of vectors to store info
  - **atomic vectors**: all entries have the same data type
  - **lists**: entries can contain other objects that can differ in data type
- All vectors have a length

```
x    # atomic vector
## [1] 8 2 1 3

length(x)
## [1] 4
```

```
x_list <- list(x, 1, "a")  # list

length(x_list)
## [1] 3
```

Atomic Vectors: Matrices

- You can add **attributes**, such as **dimension**, to vectors
- A **matrix** is a 2-dimensional vector containing entries of the same type

```
x_mat <- matrix(x, nrow = 2, byrow = TRUE)
x_mat
##      [,1] [,2]
## [1,]    8    2
## [2,]    1    3
attributes(x_mat)
## $dim
## [1] 2 2
```

```
typeof(x_mat)   # type of entries
## [1] "double"
class(x_mat)   # info about object
## [1] "matrix" "array"
```

- or you can bind vectors of the same length to create columns or rows:

```
x_mat2 <- cbind(x, 2*x)
x_mat2
##      x
## [1,] 8 16
## [2,] 2  4
## [3,] 1  2
## [4,] 3  6
```

Lists: Data frames

- A **data frame** is a list of atomic vectors of the same length, but not necessarily the same data type
- the `babynames` data frame has columns that are `integer` and `character` types

```
babynames <- read.csv("https://raw.githubusercontent.com/deepbas/statdatasets/main/baby-names-by-state.csv")
class(babynames)
## [1] "data.frame"
typeof(babynames)
## [1] "list"
glimpse(babynames)
## Rows: 502,618
## Columns: 5
## $ state  <chr> "AK", "AK", "AK", "AK", "AK", "AK", "AK", "AK", "AK", "AK", "AK…
## $ year   <int> 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 1960, 196…
## $ name   <chr> "David", "Michael", "Robert", "John", "James", "Mark", "Richard…
## $ number <int> 151, 139, 135, 126, 123, 91, 86, 74, 73, 66, 57, 50, 49, 43, 43…
## $ sex    <chr> "boy", "boy", "boy", "boy", "boy", "boy", "boy", "boy", "boy", …
```

Coercion and the factor class

- Entries in atomic vectors must be the same data type
- R will default to the most complex data type if more than one type is given

```
y <- c(1, 2, "a")
```

```
typeof(y)
## [1] "character"
y
## [1] "1" "2" "a"
```

- This example was **implicit coercion**
- **Explicit coercion** intentionally forces a data type that is different from the "default" type

```
y <- as.character(c(1,2,3))
```

```
typeof(y)
## [1] "character"
y
## [1] "1" "2" "3"
```

Coercion and the factor class

- Logical values coerced into 0 for `FALSE` and 1 for `TRUE` when applying math functions

```
x <- c(8,2,1,3)
x >= 5  # which entries >= 5?
## [1]  TRUE FALSE FALSE FALSE
sum(x >= 5)  # how many >=5 ?
## [1] 1
```

- What will `mean` of a logical vector measure?

```
mean(x >= 5)
```

```
## [1] 0.25
```

Data types: factors

- Factors are a class of data that are stored as **integers**

```
x_fct <- as.factor(c("yes", "no", "no"))

class(x_fct)
## [1] "factor"
typeof(x_fct)
## [1] "integer"
```

- The attribute `levels` is a character vector of possible values
  - Values are stored as the integers (1=first `level`, 2=second `level`, etc)
  - Levels are ordered alphabetically/numerically (unless specified otherwise)

```
str(x_fct)
##  Factor w/ 2 levels "no","yes": 2 1 1
levels(x_fct)
## [1] "no"  "yes"
```

Subsetting: Atomic Vector

- subset with `[]` by referencing index value (from 1 to vector length):

```
x
## [1] 8 2 1 3
x[c(4, 2)]  # get 4th and 2nd entries
## [1] 3 2
```

- subset by omitting entries

```
x[-c(4, 2)]  # omit 4th and 2nd entries
## [1] 8 1
```

- subset with a logical vector

```
x[c(TRUE, FALSE, TRUE, FALSE)]  # get 1st and 3rd entries
## [1] 8 1
```

Subsetting: Matrices

- access entries using subsetting `[row,column]`

```
x_mat2
##      x
## [1,] 8 16
## [2,] 2  4
## [3,] 1  2
## [4,] 3  6
```

```
x_mat2[, 1] # first column
## [1] 8 2 1 3
```

```
x_mat2[1:2, 1] # first 2 rows of first column
## [1] 8 2
```

- R doesn't always preserve class:

```
class(x_mat2[1,])  # one row (or col) is no longer a matrix (1D)
## [1] "numeric"
```

Subsetting: Data frames

- you can access entries like a matrix:

```
x_df <- data.frame(x = x, double_x = x*2)
x_df
##   x double_x
## 1 8       16
## 2 2        4
## 3 1        2
## 4 3        6
```

```
x_df[, 1]  # first column, all rows
## [1] 8 2 1 3
```

```
class(x_df[, 1])  # first column is no longer a data frame
## [1] "numeric"
```

- One column of a data frame is no longer a data frame

Subsetting: Data frames

- or access columns with `$`

```
x_df$x  # get variable x column
## [1] 8 2 1 3
```

- you can also use column names to subset:

```
# get 2 rows of Name and Sex
babynames[1:2, c("name", "sex")]
##      name sex
## 1   David boy
## 2 Michael boy
```

- Recall: a **list** is a vector with entries that can be different object types

```
my_list <- list(myVec = x,
                myDf = x_df,
                myString = c("hi", "bye"))
my_list
## $myVec
## [1] 8 2 1 3
##
## $myDf
##   x double_x
## 1 8       16
## 2 2        4
## 3 1        2
## 4 3        6
##
## $myString
## [1] "hi"  "bye"
```

Subsetting: Lists

- Like a data frame, can use the `$` to access **named** objects stored in the list
  - E.g.: `my_list$myDf` return the **data frame** `myDf`

```
my_list$myDf
##   x double_x
## 1 8       16
## 2 2        4
## 3 1        2
## 4 3        6
class(my_list$myDf)
## [1] "data.frame"
```

- one `[]` operator gives you the object at the given location but preserves the list type
  - `my_list[2]` return a **list** of length one with entry `myDf`

```
my_list[2]
## $myDf
##   x double_x
## 1 8       16
## 2 2        4
## 3 1        2
## 4 3        6
```

```
str(my_list[2])
## List of 1
##  $ myDf:'data.frame':    4 obs. of  2 variables:
##   ..$ x       : num [1:4] 8 2 1 3
##   ..$ double_x: num [1:4] 16 4 2 6
```

Subsetting: Lists

- the double `[[]]` operator gives you the object stored at that location (equivalent to using `$`)
  - `my_list[[2]]` or `my_list[["myDf"]]` return the **data frame** `myDf`

```
my_list[[2]]
##   x double_x
## 1 8       16
## 2 2        4
## 3 1        2
## 4 3        6
str(my_list[[2]])
## 'data.frame':    4 obs. of  2 variables:
##  $ x       : num  8 2 1 3
##  $ double_x: num  16 4 2 6
```

Your Turn

Please git clone the repository 02-Data-Objects-Viz to your local folder.

```
```{r}
babynames <- read.csv("https://raw.githubusercontent.com/deepbas/statdatasets/main/baby-names-by-state.csv")
x <- c(3,6,9,5,1)
x.mat <- cbind(x, 2*x)
x.df <- data.frame(x=x,double.x=x*2)
my.list <- list(myVec=x, myDf=x.df, myString=c("hi","bye"))

```
```

Question 1: data types

- What data type is `x`? What data type is `babynames$number`?
- What data type is `c(x, babynames$year)`?
- What data type is `c(x,NA)`? What data type is `c(x,"NA")`?

03:00

Your Turn

Question 2: Subsetting and coercion

- How can we reverse the order of entries in `x`?
- What does `which(x < 5)` equal?
- What does `sum(c(TRUE,FALSE,FALSE,FALSE, TRUE))` equal?
- What does `sum(x[c(TRUE,FALSE,FALSE,FALSE, TRUE)])` equal?
- What does `sum(x < 5)` equal?
- What does `sum(x[x < 5])` equal?
- Why `dim(x.mat[1:2,1])` return `NULL` while `dim(x.mat[1:2,1:2])` returns a dimension?

05:00

Your Turn

Question 3: Lists

- Using `my.list`, show three ways to write one command that gives the 3rd entry of variable `x` in data frame `myDf`
- What class of object does the command `my.list[3]` return?
- What class of object does the command `my.list[[3]]` return?
- What class of object does the command `unlist(my.list)` return? Why are all the entries `character`s?

Question 4: Data Frames

- What is the total number of babies born in Minnesota with name `Alex`?
- In what year were highest number of babies were named `Alex` in Minnesota?

03:00

03:00

Data Visualization

provides a powerful way to communicate data-driven findings, motivate analyses, and detect flaw

ggplot2 — Overview

- A powerful package for visualising data
- Used widely by academics and industries alike
- Some useful resources
  - The package documentation
  - The book by its creator Hadley Wickham
  - The reference page
  - The extensions, maintained by the `ggplot2` community

ggplot2 — Basics

- The `ggplot` function and the `data` argument
  - specify a data frame in the main `ggplot` function

```
ggplot(data = df)
```

- The mapping aesthetics, or aes; most importantly, the variable(s) that we want to plot
  - specify as an additional argument in the same `ggplot` function

```
ggplot(data = df, mapping = aes(x = x-variable, y = y-variable))
```

ggplot2 — Basics

- The geometric objects, or geom; the visual representations
  - specify, after a plus sign `+`, as an additional function

```
ggplot(data = df, mapping = aes(x = x-variable, y = y-variable)) +
       geom_point()
```

- Additional aesthestics like `color`, `size`, `shape`, and `alpha` (i.e. transparency) are possible.

```
ggplot(data = df, mapping = aes(x = x-variable, y = y-variable)) +
       geom_point(color = z-variable)
```

Your Turn

```
#install.packages("tidyverse")
library(tidyverse)
babynames_MN_John <- babynames %>% filter(state=="MN",  name == "John")
```

Question 5: Basic Plot using `ggplot2`.

- What are the grammar of graphics needed to create a scatter-plot of the number of babies born in Minnesota named `John` vs year they were born?
- Fill in the data and aesthetic mapping in the below code chunk. What is returned? What's missing?
- Add the appropriate geometric object to create the scatter plot. This is called adding a *layer* to a plot.
- Repeat the above steps with babies named `John` from Minnesota or Michigan.

03:00

Your Turn — add appropriate labels

```r
ggplot(data = babynames_MN_MI_John , mapping = aes(x = year, y = number)) +
  geom_point(aes(colour=state))+
  xlab("Year") + ylab(bquote(Number~of~babies~named~.("John"))  +
  theme(plot.title = element_text(hjust = 0.5))    # center the plot title
```