How R thinks about data

- Be able to describe the different data types R uses
- Use c(), str(), class(), and typeof() to make and investigate vectors
- Understand coercion between data types
- Know how factors work under the hood
- Be able to manipulate factors

Why?

- R is like speaking another language so its important to understand what you are telling R to do
- learning about how R thinks is fundamental to this
- really important to know because it will help you trouble shoot error message & deal with problems are that are hard to Google

Vectors

- most basic way R deals with about data
- any series of values (numbers or text)
- we assign these values to an *object*
- bind values together with c() function

```
weight_g <- c(50, 60, 65, 82)
weight_g
```

```
## [1] 50 60 65 82
```

```
animals <- c("mouse", "rat", "dog")
animals
```

```
## [1] "mouse" "rat"   "dog"
```

```
# inspection functions
length(weight_g)
```

```
## [1] 4
```

```
length(animals)
```

```
## [1] 3
```

```
class(weight_g)
```

```
## [1] "numeric"
```

```
class(animals)
```

```
## [1] "character"
```

```
str(weight_g)
```

```
##  num [1:4] 50 60 65 82
```

```r
str(animals)
```

```
##  chr [1:3] "mouse" "rat" "dog"
```

```r
# change vectors
weight_g <- c(weight_g, 90) # add to the end of the vector

weight_g <- c(30, weight_g) # add to the beginning of the vector weight_g
```

Challenge

- We've seen that atomic vectors can be of type character, numeric (or double), integer, and logical. But what happens if we try to mix these types in a single vector?
- What will happen in each of these examples? (hint: use `class()` to check the data type of your objects):

```r
num_char <- c(1, 2, 3, "a")
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
tricky <- c(1, 2, 3, "4")
```

- Why do you think it happens?
- How many values in `combined_logical` are `"TRUE"` (as a character) in the following example:

```r
num_logical <- c(1, 2, 3, TRUE)
char_logical <- c("a", "b", "c", TRUE)
combined_logical <- c(num_logical, char_logical)
```

- You've probably noticed that objects of different types get converted into a single, shared type within a vector. In R, we call converting objects from one class into another class *coercion*. These conversions happen according to a hierarchy, whereby some types get preferentially coerced into other types. Can you draw a diagram that represents the hierarchy of how these data types are coerced?

Subsetting

- If we want to extract one or several values from a vector, we must provide one or several indices in square brackets.

  ```r
  animals <- c("mouse", "rat", "dog", "cat")
  animals[2] # could be read as "return the second value in animals"
  ```

  ```
  ## [1] "rat"
  ```

  ```r
  animals[c(3, 2)] # could be read as "return the third and second values in animals" weight_g
  ```

  ```
  ## [1] "dog" "rat"
  ```

  ```r
  #We can also repeat the indices to create an object with more elements than the original one:
  animals[c(1, 2, 3, 2, 1, 4)]
  ```

  ```
  ## [1] "mouse" "rat"   "dog"   "rat"   "mouse" "cat"
  ```

Conditional subsetting —-

- Another common way of subsetting is by using a logical vector. `TRUE` will select the element with the same index, while `FALSE` will not:
- logical vectors are usually an intermediate step in subsetting

```r
weight_g <- c(21, 34, 39, 54, 55)
weight_g[c(TRUE, FALSE, TRUE, TRUE, FALSE)] # could be read as "give me the first value, not the second
```

```
## [1] 21 39 54
```

```r
weight_g > 50 # will return logicals with TRUE for the indices that meet the condition
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE
```

```r
# if we want to actually select values about 50
weight_g[weight_g > 50]
```

```
## [1] 54 55
```

```r
#You can combine multiple tests using & (both conditions are true, AND) or | (at least one of the condi
weight_g[weight_g < 30 | weight_g > 50]
```

```
## [1] 21 54 55
```

```r
## different symbols
# >= greater than
# =< less than
# == equal
# %in% within

# Example
animals <- c("mouse", "rat", "dog", "cat")
animals[animals == "cat" | animals == "rat"] # returns both rat and cat
```

```
## [1] "rat" "cat"
```

```r
animals
```

```
## [1] "mouse" "rat"   "dog"   "cat"
```

```r
# VS
# see what values in a specific list are within a bigger list
animals %in% c("rat", "cat", "dog", "duck", "goat")
```

```
## [1] FALSE  TRUE  TRUE  TRUE
```

```r
animals[animals %in% c("rat", "cat", "dog", "duck", "goat")]
```

```
## [1] "rat" "dog" "cat"
```

```r
## Challenge
# why does this return TRUE?
"four" > "five"
```

```
## [1] TRUE
```

Vector Math

- You can add a number to a vector of numbers like this:

```r
x <- 1:10
x + 3
```

```
##  [1]  4  5  6  7  8  9 10 11 12 13
```

```r
x * 10
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

```
# adding two vectors together of the SAME length
y <- 100:109
x + y
```

```
##  [1] 101 103 105 107 109 111 113 115 117 119
```

```
# different length -- RECYCLING
z <- 1:2
x + z
```

```
##  [1]  2  4  4  6  6  8  8 10 10 12
```

```
#Whoa... what happened here? R does something called recycling. It adds together the first values of ea

# save as a new object
a <- x + z
# R warns us about this! However, if you try to assign this result to an object, we get the warning, bu
```

Missing data

- When doing operations on numbers, most functions will return `NA` if the data you are working with include missing values.
- You can add the argument na.rm=TRUE to calculate the result while ignoring the missing values

```
heights <- c(2, 4, 4, NA, 6)
mean(heights)
```

```
## [1] NA
```

```
mean(heights, na.rm = TRUE)
```

```
## [1] 4
```

```
#Extract those elements which are not missing values.
is.na(heights) # this returns a logical vector with TRUE where there is an NA
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE
```

```
!is.na(heights) # the ! means "is not", so now we get a logical vector with FALSE for NAs
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE
```

```
heights[!is.na(heights)] # now we put that logical vector in, and it will NOT return the entries with N
```

```
## [1] 2 4 4 6
```

```
#Extract those elements which are complete cases. The returned object is an atomic vector of type "nume
heights[complete.cases(heights)]
```

```
## [1] 2 4 4 6
```

Challenge

1. Using this vector of heights in inches, create a new vector with the NAs removed.

```
heights <- c(63, 69, 60, 65, NA, 68, 61, 70, 61, 59, 64, 69, 63, 63, NA, 72, 65, 64, 70, 63, 65)
```

2. Use the function `median()` to calculate the median of the `heights` vector.
3. Use R to figure out how many people in the set are taller than 67 inches.

Other Data Structures

Vectors are one of the many **data structures** that R uses. Other important ones are lists (`list`), data frames (`data.frame`), matrices (`matrix`), arrays (`array`), and factors (`factor`). These are all built from combinations of vectors, so much of what you learned about vectors will be important when working with these data structures.

- lists: multiple vectors, dif data types
- data frame: most common, basically just a picky list that has to have the same length, multiple data types
- matrices and arrays: single type of data, not super common
- factors: fancier vector

Why factors

- Factors can be convenient at times, and they will pop up pretty frequently, but in most circumstances, character strings will give you fewer hassles.
- It's usually best to start with character vectors, and convert them explicitly to factors if you need to.
- Some functions in R will automatically convert character strings to factors. For instance, read.csv() run in older versions R will turn any character data into factors, while in newer versions this has been changed to keep them as characters.
- If you aren't sure, you can use the argument stringsAsFactors=FALSE in read.csv() to make sure your character strings as character strings.

```r
# Factors are used to represent categorical data
animals <- factor(c("duck", "duck", "goose", "goose"))
class(animals)
```

```
## [1] "factor"
```

```r
typeof(animals)
```

```
## [1] "integer"
```

```r
levels(animals)
```

```
## [1] "duck"  "goose"
```

```r
nlevels(animals)
```

```
## [1] 2
```

```r
## current order
animals
```

```
## [1] duck  duck  goose goose
## Levels: duck goose
```

```r
animals <- factor(animals, levels = c("goose", "duck"))
animals # after re-ordering
```

```
## [1] duck  duck  goose goose
## Levels: goose duck
```

Convert factors

```
as.character(animals) # returns index values of the characters
```

```
## [1] "duck"  "duck"  "goose" "goose"
```

```
# When is this an issue?
year_fct <- factor(c(1990, 1983, 1977, 1998, 1990)) #WRONG
as.numeric(year_fct)
```

```
## [1] 3 2 1 4 3
```

```
as.numeric(as.character(year_fct)) # This does the trick
```

```
## [1] 1990 1983 1977 1998 1990
```

Renaming factors

```
# rename using the levels function
levels(animals)
```

```
## [1] "goose" "duck"
```

```
# get the value
levels(animals)[1]
```

```
## [1] "goose"
```

```
# let's make it capital
levels(animals)[1] <- "GOOSE"
animals
```

```
## [1] duck  duck  GOOSE GOOSE
## Levels: GOOSE duck
```

```
# change both
levels(animals) <- c("GOOSE", "DUCK")
animals
```

```
## [1] DUCK  DUCK  GOOSE GOOSE
## Levels: GOOSE DUCK
```

Challenge

- Copy, paste and run the code below in your R script:
  `treatment <- factor(c("high", "low", "low", "medium", "high"))`
- First, re-order the levels of `treatment` so that "low" is first, "medium" is second, and "high" is third. Hint: Use the `factor()` function again, but with an additional `levels` argument.
- Next, check the names with the `levels()` function, then use this same function to rename the levels of treatment to "L", "M" and "H"

```
treatment <- factor(c("high", "low", "low", "medium", "high"))
treatment <- factor(treatment, levels = c("low", "mediam", "high"))
levels(treatment) <- c("L", "M", "H")
treatment
```

```
## [1] H    L    L    <NA> H
## Levels: L M H
```