R Basics

Data Science for Research Assistants

Our Goal

This class is a tour of the R languages, but also on how to analyze data to gain insights and perspectives

- There is a lot of information out there
- We will only cover a few of them

Learning by Doing

Agenda

- *Course overview*
- Load and export data
- R essentials including Built-in data types and operations
- Data structures: Vectors, arrays, lists, and data frames
- Addressing data
- R markdown

Part I

R essentials

Questions

- What is R?
- What is Rstudio?
- Where is the inputs and outputs in Rstudio?
- How do I read data in R?
- Where can I see my data?
- What are function?
- What are the different data types in R?

Keypoints

- R and Rstudio elements: source, console, packages, files and outputs
- R's basic data types are character, numeric, integer, complex, and logical.
- Functions to check data types
- Unary and Binary operators

Objectives

- Familiarize you with R and Rstudio user interface
- Expose you to the different data types in R.
- Learn different types of operators
- Be able to check the type of vector.
- Learn about missing data and other special values.

What is R/RStudio?

- R is a statistical programming language
- RStudio is a convenient interface for R (an integreated development environment, IDE)

Let's Take a tour in Rstudio

R essentials

- **Data** are things like 7, “seven”, 7.000, and

$$\begin{bmatrix} 7 & 7 & 7 \\ 7 & 7 & 7 \end{bmatrix}$$

- **Functions** are (most often) verbs, followed by what they will be applied to in parantheses:

```
do_this(to_this)
do_that(to_this, to_that, with_those,
return_something)
```

R essentials

- **Packages** are installed with the `install.packages` function and loaded with the `library` function, once per session:

```
install.packages("package_name")
library(package_name)
```

- Now Try it yourself

```
install.packages("tidyverse")
```

```
library(tidyverse)
```

What is the Tidyverse?

tidy

- The tidyverse is an opinionated collection of R packages designed for data science with similar underlying philosophy and a common syntax.

Loading Starwars data

```
library(tidyverse)
starwars
```

Lets take a look at this data.

Loading data from files

Let's import the file called `strike.csv` into our R environment. To import the file we need to tell our computer where the file is. We do that by choosing a *working directory*, that is, a local directory on our computer containing the files we need.

For this example, we change the path to our new directory at where the lecture slides located. For example:

```
setwd("~/Downloads/R_basics/") # Mac
setwd("C:/Users/Name/Downloads/R_basics") #
Windows
```

Those who are using server based Rstudio, please upload the file, `strike.csv`.

Alternatively you can change the working directory using the RStudio GUI using the menu option `Session` -> `Set Working Directory` -> `Choose Directory...`

Loading data from files

The data file is located inside the working directory. Now we can load the data into R using `read.csv`:

```
strike_df<-read.csv(file = "strike.csv")
```

The expression `read.csv(...)` is a *function call* that asks R to run the function `read.csv`.

`read.csv` has one *arguments*: the name of the file we want to read.

Assign values to variables

We can think of a variable as a container with a name, such as `x`, `current_temperature`, or `subject_id` that contains one or more values. We can create a new variable and assign a value to it using `<-`.

```
weight_kg <- 55

weight_kg
```

We can treat our variable like a regular number, and do arithmetic with it:

```
# weight in pounds:
2.2 * weight_kg
```

We can add comments to our code using the # character. It is useful to document our code in this way so that others (and us the next time we read it) have an easier time following what the code is doing.

We can also change a variable's value by assigning it a new value:

```
weight_kg <- 57.5
# weight in kilograms is now
weight_kg
```

Note: When you assign a value to a variable, R only stores the value, not the calculation you used to create it. This is an important point if you're used to the way a spreadsheet program automatically updates linked cells.

Combine assignment and reading

Now that we know how to assign things to variables, let's re-run `read.csv` and save its result into new variable called 'dat':

```
dat<-read.csv(file = "strike.csv")
```

This statement doesn't produce any output because the assignment doesn't display anything.

Now that we've seen some data frames, let's take a look at what kinds of data we will encounter in a data frame.

R has 6 atomic data types.

- character
- numeric (real or decimal)
- integer
- logical
- missing values

By atomic,

We mean basic data

- **character**: `"a"`, `"swc"`
- **numeric**: `2`, `15.5`
- **integer**: whole numbers (positive, negative or zero) `2L` (the `L` tells R to store this as an integer)
- **logical**: direct binary values: `TRUE`, `FALSE`
- **missing values**: `NA`, `NaN`, etc.

Functions of data objects

- `class()` - what kind of object is it (high-level)?
- `typeof()` - what is the object's data type (low-level)?
- `is.`*foo*`()` functions return Booleans for whether the argument is of type *foo*
- `as.`*foo*`()` (tries to) "cast" its argument to type *foo*, to translate it sensibly into such a value

```r
y <- 10
y
class(y)

z <- as.character(y)
z
is.numeric(z)
```

Operators

- **Unary**: take just one argument. E.g., `-` for arithmetic negation, `!` for Boolean negation
- **Binary**: take two arguments. E.g., `+`, `-`, `*`, and `/`. Also, `%%` (for mod), `^` (Exponentiation).

```
-7
7 + 5
7 - 5
7 %% 5
```

Operators can return special values

`Inf` is infinity. You can have either positive or negative infinity.

```
1/0
```

`NaN` means Not a Number. It's an undefined value.

```
0/0
```

Commonly used relational operators in R

Comparison operators

These are also binary operators; they take two objects, and give back a Boolean

```
7 > 5
7 < 5
7 >= 7
```

```
7 <= 5
7 == 5
7 != 5
```

Warning: == is a comparison operator, = is not!

Logical operators

These basic ones are & (and) and | (or)

```
(5 > 7) & (6 * 7 == 42)
(5 > 7) | (6 * 7 == 42)
(5 > 7) | (6 * 7 == 42) & (0 != 0)
```

Note: The double forms && and || are different! We'll see them later

Recap

- What is R vs Rstudio
- What is a function?
- Will `is.numeric(as.character(5))` return TRUE?
- Will `(6 * 10 > 55 ) & (9 != 10-1) | (0 == 1)` return TRUE?

Part II

Data structures

Questions

- What are the different data structures in R?
- How do I access parts within the various data structures?

Keypoints

- R's basic data structures include the vector, list, matrix, and data frame
- Objects may have attributes, such as name, dimension, and class.

Objectives

- Understand the three different ways R can address data inside a data structure
- Learn how to create vectors of different types.
- Be able to check the type of vector.
- Getting familiar with the different data structures (lists, matrices, data frames).
- Combine different methods for addressing data with the assignment operator to update subsets of data.

Data structures

R has many **data structures**. These include

- atomic vector
- list
- matrix
- data frame

Data Frame

A data frame is a very important data type in R. It's pretty much the *de facto* data structure for most tabular data and what we use for statistics.

A data frame is a *special type of list* where every element of the list has same length (i.e. data frame is a "rectangular" list).

Data frames can have additional attributes such as `rownames()`, which can be useful for annotating data, like `subject_id` or `sample_id`. But most of the time they are not used.

Some additional information on data frames

- Usually created by `read.csv()` and `read.table()`, i.e. when importing the data into R.
- Can also create a new data frame with `data.frame()` function.
- Find the number of rows and columns with `nrow(dat)` and `ncol(dat)`, respectively.

Creating Data Frames by Hand

To create data frames by hand:

```
dat <- data.frame(id = letters[1:10], x =
1:10, y = 11:20)
dat
```

Objects Attributes

Objects can have **attributes**. Attributes are part of the object. These include:

- names
- dimnames
- dim
- class
- attributes (contain metadata)

Useful Data Frame Functions

- `head()` - shows first 6 rows
- `tail()` - shows last 6 rows
- `dim()` - returns the dimensions of data frame (i.e. number of rows and number of columns)
- `nrow()` - number of rows
- `ncol()` - number of columns
- `str()` - structure of data frame - name, type and preview of data in each column

Useful Data Frame Functions

- `names()` or `colnames()` - both show the `names` attribute for a data frame
- `na.omit(dataframe)` - removes the missing values in data frame We'll see many more functions that could apply to data frame later
- `sapply(dataframe, class)` - shows the class of each column in the data frame. `sapply` is a really useful function to work with data frame by each column.

```
class(starwars)
```

R has loaded the contents into a variable called `starwars` which is a tibble, a special type of `data frame`. We will see more of this type of data frame later.

```
dim(starwars)
```

The data has 87 rows and 13 columns.

```
head(starwars)
```

First 6 rows

```
sapply(starwars,class)
```

Class of each column

Addressing Data

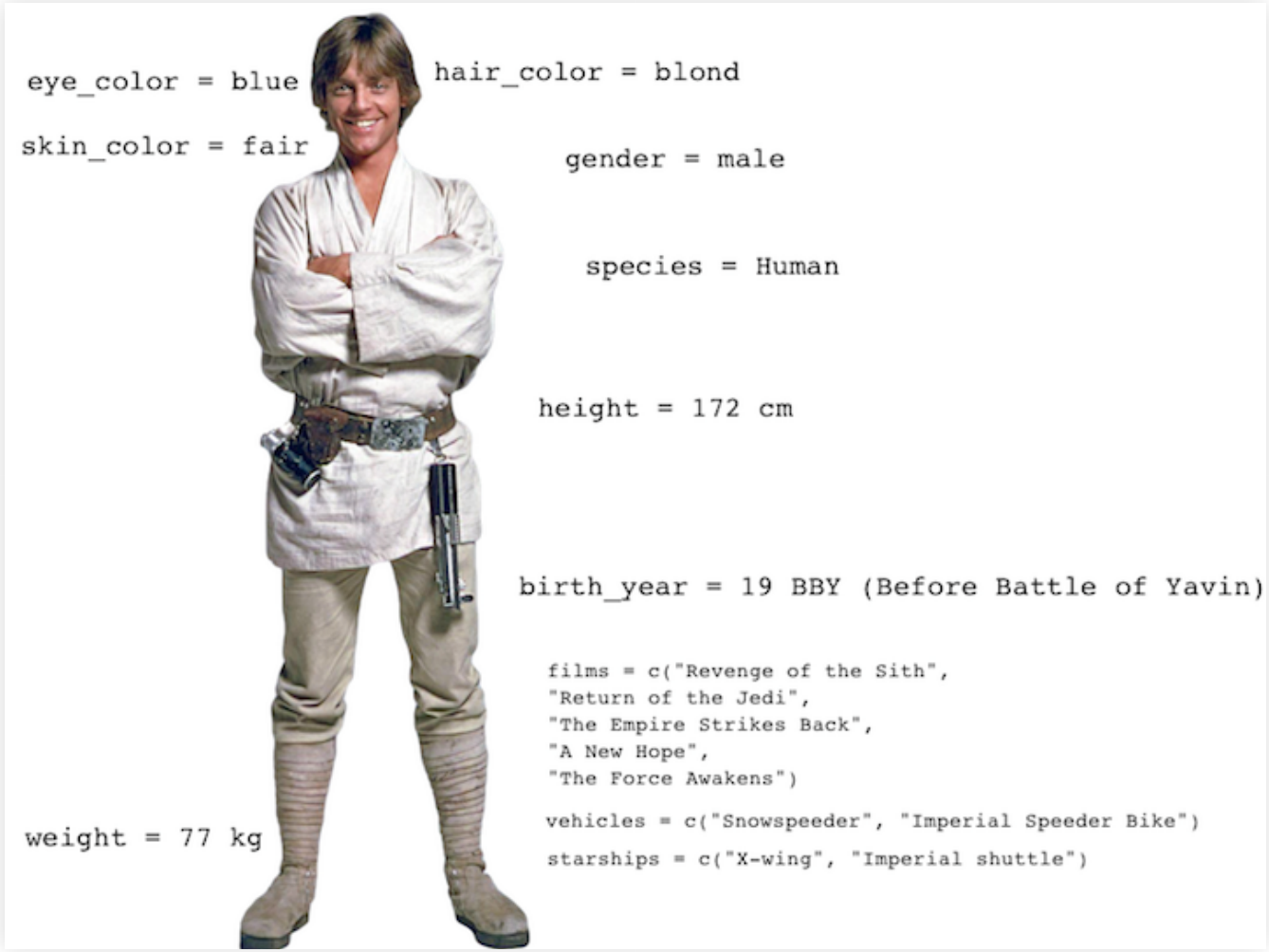
R is a powerful language for data manipulation. There are three main ways for addressing data inside R objects.

- By index (slicing)
- By logical vector
- By name (columns only)

Addressing by Index

Data can be accessed by index. The generic format is `data_frame[row_numbers,column_numbers]`.

- What will be returned by `starwars[1, 1]`?



If we leave out a dimension R will interpret this as a request for all values in that dimension.

What will be returned by `starwarss[, 2]`?

The colon `:` can be used to create a sequence of integers.

```
4:6
```

Creates a vector of numbers from 4 to 6. This can be very useful for addressing data.

Use the colon operator to index just the hair color, skin color, and eye color (columns 4 to 6).

Finally we can use the `c()` (combine) function to address non-sequential rows and columns.

```
starwars[c(1, 5, 7, 9), 1:5]
```

Returns the first 5 columns for characters in rows 1, 5, 7 & 9

Now try to return the height(column 2) and gender(column 8) values for the first 5 character.

Addressing by Name

Columns in an R data frame are named.

```
colnames(starwars)
```

We usually use the $ operator to address a column by name

```
starwars$sex
```

Addressing by Name

Named addressing can also be used in square brackets.

```
head(starwars[, c('name', 'sex')])
```

Best Practice

Best practice is to address columns by name, often you will create or delete columns and the column position will change.

Logical Indexing

A logical vector contains only the special values TRUE & FALSE. We will talk about vector next.

```
c(TRUE, TRUE, FALSE, FALSE, TRUE)
```

Note the values TRUE and FALSE are all capital letters and are not quoted.

Logical vectors can be created using `relational operators` e.g. `<, >, ==, !=, %in%`.

```
x <- c(1, 2, 3, 11, 12, 13)
x < 10

x %in% 1:10
```

Combining Indexing and Assignment

The assignment operator **<–** can be combined with indexing.

```
x <- c(1, 2, 3, 11, 12, 13)
x[x < 10] <- 0
x
```

We can use logical vectors to select data from a data frame.

```
index <- starwars$species == 'Human'
starwars[index,]$name
```

Often this operation is written as one line of code:

```
print(starwars[starwars$species ==
'Human',]$name)
```

1. Create a scatterplot using `plot()` showing Height for human characters by changing `print` to `plot`

Atomic Vectors

A vector is the most common and basic data structure in R and is pretty much the workhorse of R. Technically, vectors can be one of two types:

- atomic vectors
- lists

although the term “vector” most commonly refers to the atomic types not to lists.

The Different Vector Modes

A vector is a collection of elements that are most commonly of mode `character`, `logical`, `integer` or `numeric`.

You can create an empty vector with `vector()`. (By default the mode is `logical`.)

```
vector() # an empty 'logical' (the default) vector
vector("character", length = 5) # a vector of mode 'character' with 5 elements
```

You can be more explicit It is more common to use direct constructors such as `character()`, `numeric()`, etc.

```
character(5) # the same thing, but using the
constructor directly
numeric(5)   # a numeric vector with 5
elements
logical(5)   # a logical vector with 5
elements
```

You can also create vectors by directly specifying their content. R will then guess the appropriate mode of storage for the vector. For instance:

```
x <- c(1, 2, 3)
```

will create a vector `x` of mode `numeric`. These are the most common kind, and are treated as double precision real numbers.

If you wanted to explicitly create integers, you need to add an `L` to each element (or *coerce* to the integer type using `as.integer()`).

```
x1 <- c(1L, 2L, 3L)
identical(x1, as.integer(x))
```

Using `TRUE` and `FALSE` will create a vector of mode `logical`:

```
y <- c(TRUE, TRUE, FALSE, FALSE)
```

While using quoted text will create a vector of mode `character`:

```
z <- c("Sarah", "Tracy", "Jon")
```

Examining Vectors

The functions `typeof()`, `length()`, `class()` and `str()` provide useful information about your vectors and R objects in general.

```
typeof(z)
length(z)
class(z)
str(z)
```

Do you see a property that's common to all these vectors above?

Run this and see if you can spot the commonality

```
x
x1
y
z
```

Adding Elements

The function `c()` (for combine) can also be used to add elements to a vector.

```
z <- c(z, "Annette")
z
z <- c("Greg", z)
z
```

You can create vectors as a sequence of numbers.

```r
series <- 1:10
seq(10)
seq(from = 1, to = 10, by = 0.1)
```

Missing Data

R supports missing data in vectors. They are represented as **NA** (Not Available):

```
x <- c(0.5, NA, 0.7)
x <- c(TRUE, FALSE, NA)
x <- c("a", NA, "c", "d", "e")
```

The function `is.na()` indicates the elements of the vectors that represent missing data, and the function `anyNA()` returns `TRUE` if the vector contains any missing values:

```
x <- c("a", NA, "c", "d", NA)
is.na(x)
anyNA(x)
```

What Happens When You Mix Types Inside a Vector?

R will create a resulting vector with a mode that can most easily accommodate all the elements it contains. This conversion between modes of storage is called “coercion”. When R converts the mode of storage based on its content, it is referred to as “implicit coercion”. For instance, can you guess what the following do (without running them first)?

```r
xx <- c(1.7, "a")
xx <- c(TRUE, 2)
xx <- c("a", TRUE)
```

You can also control how vectors are coerced explicitly using the `as.<class_name>()` functions:

```r
as.numeric(c("1", "2", "3"))
as.character(1:2)
```

You can also glean other attribute-like information such as length (works on vectors and lists) or number of characters (for character strings).

```
length(1:10)
nchar("This is not a sentence")
```

List

In R lists act as containers. Unlike atomic vectors, the contents of a list are not restricted to a single mode and can encompass any mixture of data types. Lists are sometimes called generic vectors, because the elements of a list can by of any type of R object, even lists containing further lists. This property makes them fundamentally different from atomic vectors.

A list is a special type of vector. Each element can be a different type.

Create lists using `list()` or coerce other objects using `as.list()`. An empty list of the required length can be created using `vector()`

```
x <- list(1, "a", TRUE, 1+4i)
x

x <- vector("list", length = 5) # empty list
length(x)
```

The content of elements of a list can be retrieved by using double square brackets.

```
x[[1]]
```

Vectors can be coerced to lists as follows:

```
x <- 1:10
x <- as.list(x)
length(x)
```

1. What is the class of `x[1]`?
2. What about `x[[1]]`?

Elements of a list can be named (i.e. lists can have the `names` attribute)

```
my.pie = list(type="key lime", diametter=7,
is.vegetarian=TRUE)
my.pie
names(my.pie)
```

A list does not print to the console like a vector. Instead, each element of the list starts on a new line.

Elements are indexed by double brackets `[[]]`. Single brackets `[]` will still return a(nother) list. If the elements of a list are named, they can be referenced by the `$` notation (i.e. `my.pie$type`).

See that a data frame is actually a special list:

```
is.list(dat)
class(dat)
```

As data frames are also lists, it is possible to refer to columns (which are elements of such list) using the list notation, i.e. either double square brackets or a $.

```
dat[["y"]]
dat$y
```

Matrix

In R matrices are an extension of the numeric or character vectors. They are not a separate type of object but simply an atomic vector with dimensions; the number of rows and columns.

```
m <- matrix(nrow = 2, ncol = 2)
m
dim(m)
```

You can check that matrices are vectors with a class attribute of `matrix` by using `class()` and `typeof()`.

```
m <- matrix(c(1:3))
class(m)
typeof(m)
```

While `class()` shows that m is a matrix, `typeof()` shows that fundamentally the matrix is an integer vector.

Matrices in R are filled column-wise.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
```

Other ways to construct a matrix

```
m       <- 1:10
dim(m) <- c(2, 5)
```

This takes a vector and transforms it into a matrix with 2 rows and 5 columns.

The following table summarizes the one-dimensional and two-dimensional data structures in R in relation to diversity of data types they can contain.

Dimensions	Homogenous	Heterogeneous
1-D	atomic vector	list
2-D	matrix	data frame