# Coursera - Data Science & R Course

Daniel O.

15/12/2020

## Types of Data Science Questions

### Descriptive

**Goal:** To describe or summarize a set of data

- Early analysis when receiving new data
- Generate simple summaries about the samples and their measurements
- **Not** for generalizing the results of the analysis to a larger population

### Exploratory

**Goal:** To examine the data and find relationships that weren't previously known

- Explore how different variables might be related
- Useful for discovering new connections
- Help to formulate hypotheses and drive the design of future studies and data collection

### Inferential

**Goal:** Use a relatively small sample of data to say something about the population at large

- Provide your estimate of the variable for the population and provide your uncertainty about your estimate
- Ability to accurately infer information about the larger population depends heavily on sampling scheme

### Predictive

**Goal:** Use current and historical data to make predictions about future data

- Accuracy in predictions is dependent on measuring the right variables
- Many ways to build up prediction models with some being better or worse for specific cases

### Causal

**Goal:** See what happens to one variable when we manipulate another variable

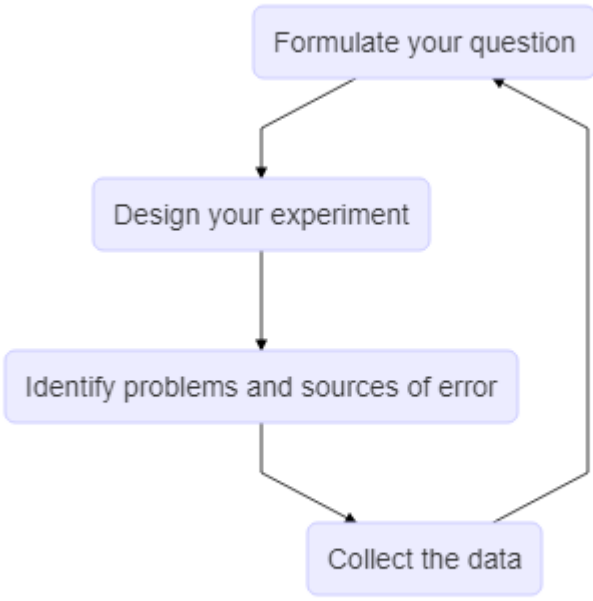
- Gold standard in data analysis
- Often applied to the results on randomized studies that were designed to identify causation
- Usually analyzed in aggregate and observed relationships are usually average effects

### Mechanistic

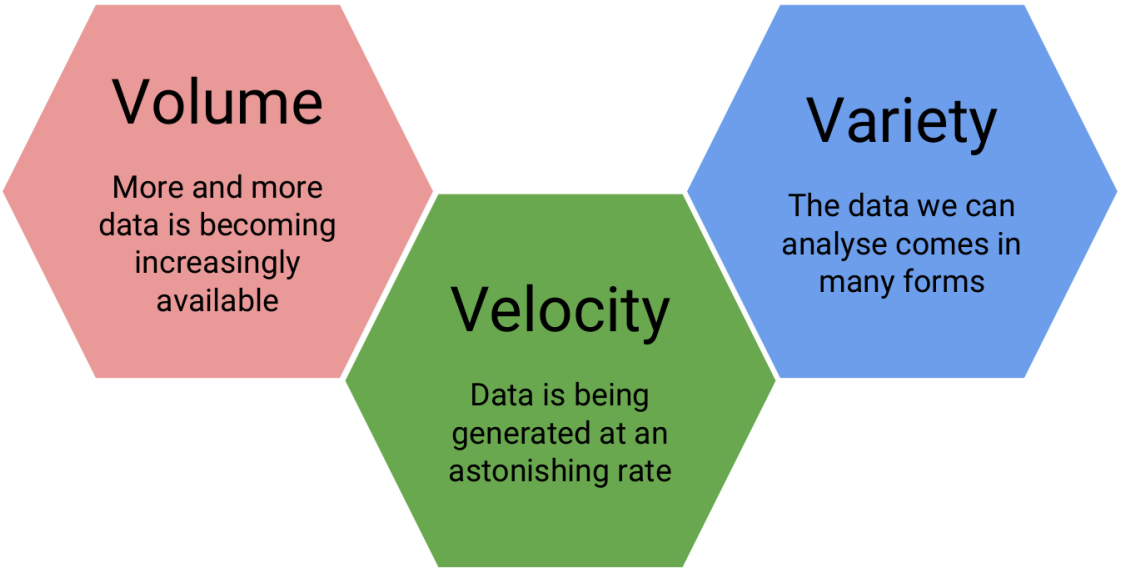
**Goal:** Understand the exact changes in variables that lead to exact changes in other variables

- Applied to simple situations or those that are nicely modeled by deterministic equations
- Commonly applied to physical or engineering sciences
- Often, the only noise in the data is measurement error

## Experimental design



## Big Data




## Getting started with R

### Directories

- getwd()
- setwd()
- dir()
- dir.create()
- file.create()
- file.exists()
- file.infor()
- file.path()
- file.rename()
- file.copy()

## Looking at data

`ls`: List the variables in the environment

`nrow` & `ncol`: Number of rows/columns in a dataframe (`dim` for both)

`object.size`: How much space the object occupies in memory

`names`: Names of columns (variables)

`head` & `tail`: Preview the top/bottom of the dataset

`summary`: Provides different output for each variable, depending on its class. For **numeric** data summary() displays the minimum, 1st quartile, median, mean, 3rd quartile, and maximum. These values help us understand how the data are distributed. For **categorical** variables (called 'factor' variables in R), summary() displays the number of times each value (or 'level') occurs in the data.

`str`: Shows the structure of the data.

## Control Structures - if/else

if(condition) {

```
  <do something>
```

} else("condition2") {

```
  <do something else>
```

}

–

if(condition) {

```
  <do something>
```

{ else if(condition2) {

```
  <do something different>
```

} else {

```
  <do something else>
```

}

## Control Structures - for loop

for(i in vect){

```
  <function>[i]
```

}

### Control Structures - while

while(finite condition){

```
  <do something>
```

}

### Control Structures - repeat, next, break, return

`repeat` is a construct that basically initiates an infinite loop. The only way to exit a `repeat` loop is to call `break`

`next` is basically used in any time of looping construct when you want to skip an iteration.

`return` signals that a function should exit and and return a given value

## Functions

Basic format:

f <- function(x, y =

){

x+y

}

User-defined binary operators have the following syntax:

```
  %[whatever]%
```

where [whatever] represents any valid variable name.

## Dates and Time

Dates are represented by the `Date` class and times are represented by the `POSIXct` and `POSIXlt` classes. Internally, dates are stored as the number of days since 1970-01-01 and times are stored as either the number of seconds since 1970-01-01 (for `POSIXct`) or a list of seconds, minutes, hours, etc. (for `POSIXlt`).

`strptime()` converts character vectors to `POSIXlt`.

If you want more control over the units when finding the above difference in times, you can use `difftime()`, which allows you to specify a 'units' parameter.

# Loop functions

**'The Split-Apply-Combine Strategy for Data Analysis'**

`lapply`: Loop over a list and evaluate a function on each element » `lapply(data, fun)`

`sapply`: Same as `lapply` but try to simplify the result

`apply`: Apply the function over the margins of an array » `apply(data, dimension, fun)`

`mapply`: Multivariate function of `lapply` *e.g.* `list(rep(1,4), rep(2,3), rep(3,2), rep(4,1))` can be done with `mapply(rep, 1:4, 4:1)`

`tapply`: Apply a function over the subsets of a vector » `tapply(data, factor, fun)`

`split`: Takes a vector and splits it according to a factor(similarly to `tapply` but without applying any type of summary statistics )

`sprintf` originally comes from C and all formatting rules are taken from it as well

*e.g.* sprintf("%03d", 7) "%03d" is a formatting string, which specifies how 7 will be printed.

d stands for decimal integer (not double!), so it says there will be no floating point or anything like that, just a regular integer.

3 shows how many digits will the printed number have. More precisely, the number will take at least 3 digits: 7 will be ___7 (with spaces instead of underscores), but 1000 will remain 1000, as there is no way to write this number with just 3 digits.

0 before 3 shows that leading spaces should be replaced by zeroes. Try experimenting with sprintf("%+3d", 7), sprintf("%-3d", 7) to see other possible modifiers (they are called flags).

# Debbuging tools

## Something's wrong

`message`: Generic notification; function execution continues

`warning`: Something went wrong but is not fatal; execution of the function is not stopped

`error`: Fatal problem has occurred; execution is halted; produced by the stop function

`condition`: A generic concept indicating that something can occur; programmers create their own conditions

## Primary Debugging Tools

`traceback`: Prints out the function call stack after an error occurs and allows to know in which function the error occurred

`recover`: Allows you to modify the error default behavior so that you can browse the function whenever an error occurs

`debug`: Flags a function for debug mode which allows you to step through the function one expression/line at a time

`browser`: Suspends the execution of a function wherever it is called and puts the function in debug mode

`trace`: Allows to insert debugging mode into a function at specific places without altering the function code

### Simulation - Generating Random Numbers

`sample`: Samples from vector

Each probability distribution in R has an r*** function (random number generation), a d*** function (density), a p*** (cumulative distribution), and q*** (quantile function).

## Getting data

download.file()

read.**file type**()

### Manipulating data with dplyr

The first step of working with data in dplyr is to load the data into what the package authors call a 'data frame tibble', using `tibble::as_tibble()` The main advantage to using a tbl_df over a regular data frame is the printing.

The dplyr philosophy is to have small functions that each do one thing well. Specifically, dplyr supplies five 'verbs' that cover most fundamental data manipulation tasks: `select()`, `filter()`, `arrange()`, `mutate()`, and `summarize()`.

#### select

Select (and optionally rename) variables (columns) in a data frame, using a concise mini-language that makes it easy to refer to variables based on their name (e.g. a:f selects all columns from a on the left to f on the right). You can also use predicate functions like is.numeric to select variables based on their properties.

#### filter

The `filter()` function is used to subset a data frame, retaining all rows that satisfy your conditions. To be retained, the row must produce a value of TRUE for all conditions. Note that when a condition evaluates to NA the row will be dropped, unlike base subsetting with [.

#### arrange

`arrange()` orders the rows of a data frame by the values of selected columns.

Unlike other dplyr verbs, `arrange()` largely ignores grouping; you need to explicitly mention grouping variables (or use .by_group = TRUE) in order to group by them, and functions of variables are evaluated once per data frame, not once per group.

#### mutate & transmute

`mutate()` adds new variables and **preserves** existing ones; `transmute()` adds new variables and **drops** existing ones. New variables overwrite existing variables of the same name. Variables can be removed by setting their value to NULL.

### summarize

`summarize()` creates a new data frame. It will have one (or more) rows for each combination of grouping variables; if there are no grouping variables, the output will have a single row summarizing all in the input. It will contain one column for each grouping variable and one column for each of the summary statistics that you have specified.

## Grouping and chaining with dplyr

### Group by one or more variables

`group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

Grouping data by variable, allows to summarize data by each value of that variable.

**e.g.**

```
library(dplyr)
data("iris")
iris_tbl <- tibble(iris)

by_species <- group_by(iris, Species)

plant_sum <- summarize(by_species,
                       count = n(),
                       unique = n_distinct(Species),
                       avg_Petal.Width = mean(Petal.Width))
```

```
plant_sum
```

```
## # A tibble: 3 x 4
##   Species    count unique avg_Petal.Width
##   <fct>      <int>  <int>           <dbl>
## 1 setosa        50      1           0.246
## 2 versicolor    50      1           1.33
## 3 virginica     50      1           2.03
```

### Chaining

Chaining makes use of the `%>%` (*pipe*) operator. With this operator, you can chain together different commands without having to provide the base data/vector argument for each one.

**e.g.**

```
myresult <-
iris_tbl %>% group_by(Species) %>%
  summarize(
              count = n(),
              unique = n_distinct(Species),
              avg_Petal.Width = mean(Petal.Width)) %>%
  filter(avg_Petal.Width > 0) %>%
  arrange(desc(avg_Petal.Width))
```

```
myresult
```

```
## # A tibble: 3 x 4
##   Species    count unique avg_Petal.Width
##   <fct>      <int>  <int>           <dbl>
## 1 virginica     50      1           2.03
## 2 versicolor    50      1           1.33
## 3 setosa        50      1           0.246
```

## Tidy data

Tidy data is formatted in a standard way that facilitates exploration and analysis and works seamlessly with other tidy data tools. Specifically, tidy data satisfies three conditions:

1. Each variable forms a column
2. Each observation forms a row
3. Each type of observational unit forms a table

### Gather columns into key-value pairs

Development on `gather()` is incomplete, and for new code we recommend switching to `pivot_longer()`, which is easier to use, more featureful, and still under active development. `df %>% gather("key", "value", x, y, z)` is equivalent to `df %>% pivot_longer(c(x, y, z), names_to = "key", values_to = "value")`.

`pivot_longer()` "lengthens" data, increasing the number of rows and decreasing the number of columns. The inverse transformation is `pivot_wider()`.

### Separate a character column into multiple columns with a regular expression or numeric locations

Given either a regular expression or a vector of character positions, `separate()` turns a single character column into multiple columns. Unless you request otherwise with the 'sep' argument, it splits on non-alphanumeric values. In other words, it assumes that the values are separated by something other than a letter or number.

### Spread a key-value pair across multiple columns

Development on `spread()` is complete, and for new code we recommend switching to `pivot_wider()`, which is easier to use, more featureful, and still under active development. `df %>% spread(key, value)` is equivalent to `df %>% pivot_wider`(names_from = key, values_from = value)'.

`pivot_wider()` "widens" data, increasing the number of columns and decreasing the number of rows. The inverse transformation is `pivot_longer()`.

### Efficiently bind multiple data frames by row and column

`bind_rows` and `bind_cols` are an efficient implementation of the common pattern of `do.call(rbind, dfs)` or `do.call(cbind, dfs)` for binding many data frames into one.

#### Getting data from the Web

The `con` command creates a connection to a URL and `readLines` can then be used to obtain the content of the page (remember to close the connection).

Data can also be obtained from a dedicated API (for Twitter, Facebook,...)

The XML package can be useful in parsing the information from the html format, **e.g.:**

```
library(XML)
url <- "https://www.researchgate.net/profile/Daniel_Oliveira69"
html <- htmlTreeParse(url,useInternalNodes=T)
xpathSApply(html,"//title",xmlValue)
```

# Exploratory Data Analysis

## Exploratory Graphs

Exploratory graphs are used...

- To understand data properties
- To find patterns in data
- To suggest modeling strategies
- To "debug" analyses

These tend to be made quickly and in large number as a way to personally understand the data and not for presentation, **e.g.:** histograms, boxplots,...

## Principles of Analytic Graphics

#### Show comparisons

Showing comparisons is really the basis of all good scientific investigation. Evidence for a hypothesis is always relative to another competing hypothesis. When you say "the evidence favors hypothesis A", what you mean to say is that "the evidence favors hypothesis A versus hypothesis B". A good scientist is always asking "Compared to What?" when confronted with a scientific claim or statement. Data graphics should generally follow this same principle. You should always be comparing at least two things.

#### Show causality, mechanism, explanation, systematic structure

If possible, it's always useful to show your causal framework for thinking about a question. Generally, it's difficult to prove that one thing causes another thing even with the most carefully collected data. But it's still often useful for your data graphics to indicate what you are thinking about in terms of cause. Such a display may suggest hypotheses or refute them, but most importantly, they will raise new questions that can be followed up with new data or analyses.

### Show multivariate data

The real world is multivariate. For anything that you might study, there are usually many attributes that you can measure. The point is that data graphics should attempt to show this information as much as possible, rather than reduce things down to one or two features that we can plot on a page. There are a variety of ways that you can show multivariate data, and you don't need to wear 3-D glasses to do it.

**NOTE** Simpson's paradox, which also goes by several other names, is a phenomenon in probability and statistics, in which *a trend appears in several different groups of data but disappears or reverses when these groups are combined*. This result is often encountered in social-science and medical-science statistics and is particularly problematic when frequency data is unduly given causal interpretations.The paradox can be resolved when causal relations are appropriately addressed in the statistical modeling. It is also referred to as Simpson's reversal, Yule–Simpson effect, amalgamation paradox, or reversal paradox.

### Integrate evidence

Just because you may be making data graphics, doesn't mean you have to rely solely on circles and lines to make your point. You can also include printed numbers, words, images, and diagrams to tell your story. In other words, data graphics should make use of many modes of data presentation simultaneously, not just the ones that are familiar to you or that the software can handle. One should never let the tools available drive the analysis; one should integrate as much evidence as possible on to a graphic as possible.

### Describe and document the evidence

Data graphics should be appropriately documented with labels, scales, and sources. A general rule for me is that a data graphic should tell a complete story all by itself. You should not have to refer to extra text or descriptions when interpreting a plot, if possible. Ideally, a plot would have all of the necessary descriptions attached to it. You might think that this level of documentation should be reserved for "final" plots as opposed to exploratory ones, but it's good to get in the habit of documenting your evidence sooner rather than later.

### Content, Content, Content

Analytical presentations ultimately stand or fall depending on the quality, relevance, and integrity of their content. This includes the question being asked and the evidence presented in favor of certain hypotheses. No amount of visualization magic or bells and whistles can make poor data, or more importantly, a poorly formed question, shine with clarity. Starting with a good question, developing a sound approach, and only presenting information that is necessary for answering that question, is essential to every data graphic.

## Plotting Systems

### Base Plotting System

It's the oldest system which uses a simple "Artist's palette" model. What this means is that you start with a blank canvas and build your plot up from there, step by step.Usually you start with a plot function (or something similar), then you use annotation functions to add to or modify your plot. R provides many annotating functions such as text, lines, points, and axis.

The base system is very intuitive and easy to use when you're starting to do exploratory graphing and looking for a research direction. You can't go backwards, though, say, if you need to readjust margins or fix a misspelled a caption. A finished plot will be a series of R commands, so it's difficult to translate a finished plot into a different system.

### Lattice System

Unlike the Base System, lattice plots are created with a single function call such as `xyplot` or `bwplot`. Margins and spacing are set automatically because the entire plot is specified at once.

The lattice system is most useful for conditioning types of plots which display how y changes with x across levels of z. The variable z might be a categorical variable of your data. This system is also good for putting many plots on a screen at once.

The lattice system has several disadvantages. First, it is sometimes awkward to specify an entire plot in a single function call. Annotating a plot may not be especially intuitive. Second, using panel functions and subscripts is somewhat difficult and requires preparation. Finally, you cannot "add" to the plot once it is created as you can with the base system.

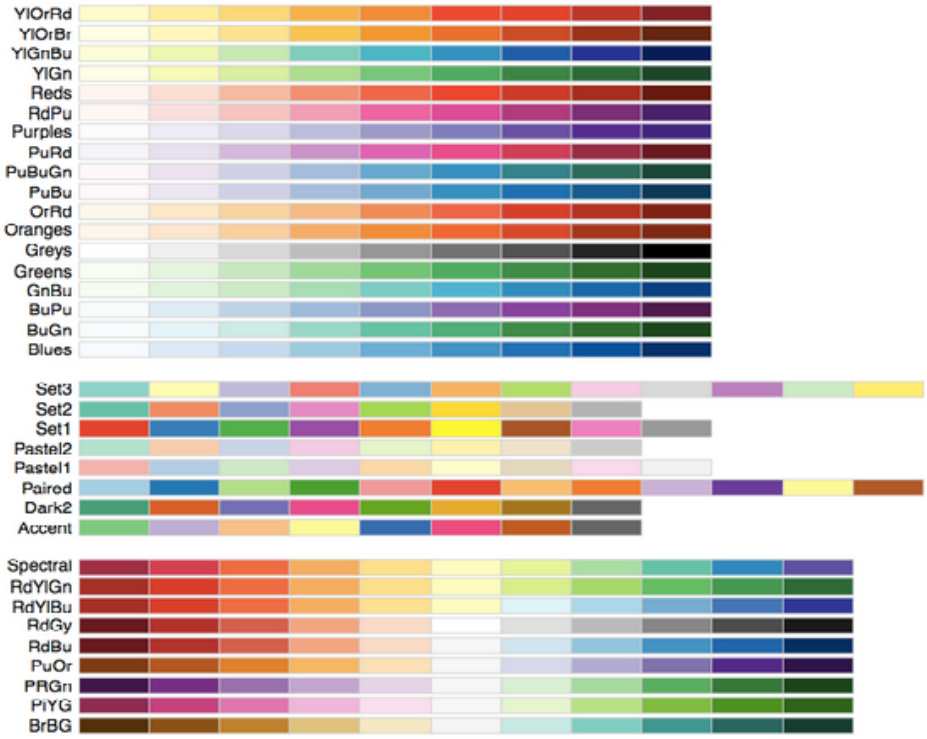
### ggplot2

ggplot2 is a hybrid of the base and lattice systems. It automatically deals with spacing, text, titles (as Lattice does) but also allows you to annotate by "adding" to a plot (as Base does), so it's the best of both worlds. Although ggplot2 bears a superficial similarity to lattice, it's generally easier and more intuitive to use. Its default mode makes many choices for you but you can still customize a lot. The package is based on a "grammar of graphics" (hence the gg in the name), so you can control the aesthetics of your plots. For instance, you can plot conditioning graphs and panel plots as we did in the lattice example.

## Working with colors

The function `colors()` lists the names of 657 predefined colors you can use in any plotting function. These names are returned as strings. So you're free to use any of these 600+ colors listed by the colors function. However, two additional functions from grDevices, `colorRamp` and `colorRampPalette`, give you more options. Both of these take color names as arguments and use them as "palettes", that is, these argument colors are blended in different proportions to form new colors. The first, `colorRamp`, takes a palette of colors (the arguments) and returns a function that takes values between 0 and 1 as arguments. The 0 and 1 correspond to the extremes of the color palette. Arguments between 0 and 1 return blends of these extremes. The second, `colorRampPalette` takes a palette of colors and returns a function. This function, however, takes integer arguments (instead of numbers between 0 and 1) and returns a vector of colors each of which is a blend of colors of the original palette.

**NOTE**: The RColorBrewer Package, available on CRAN, that contains interesting and useful color palettes, of which there are 3 types, sequential, divergent, and qualitative. Which one you would choose to use depends on your data. Here's a picture of the palettes available from this package. The top section shows the sequential palettes in which the colors are ordered from light to dark. The divergent palettes are at the bottom. Here the neutral color (white) is in the center, and as you move from the middle to the two ends of each palette, the colors increase in intensity. The middle display shows the qualitative palettes which look like collections of random colors. These might be used to distinguish factors in your data.


YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues
Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent
Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG