# DATA SOCIETY®

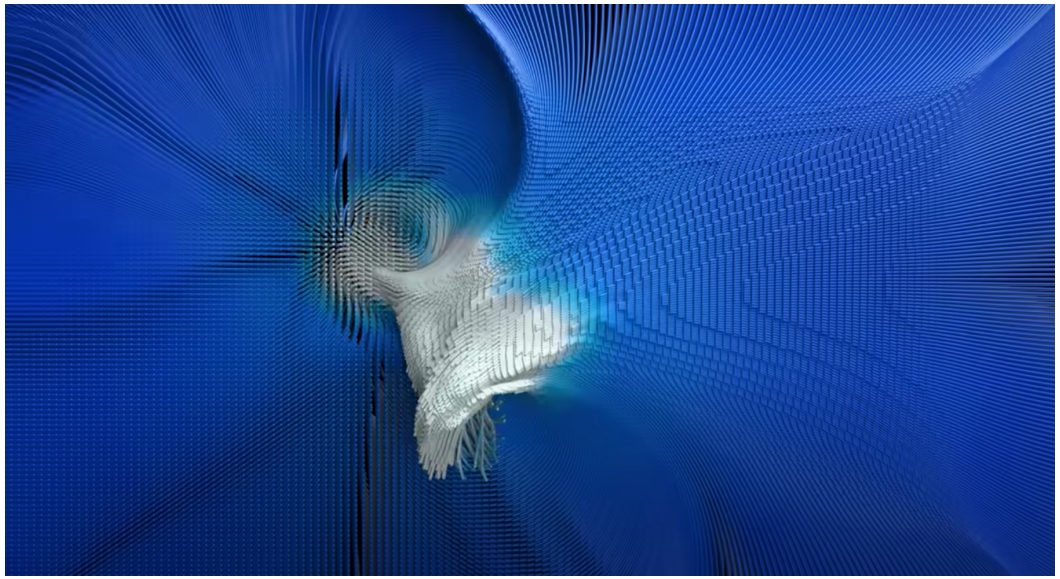
## Intro to R and visualization - Part 4

*"One should look for what is and not what he thinks should be."*
*-Albert Einstein.*

# Welcome back

- In the **last module**, we introduced the `tidyverse` and `dplyr` packages and used them to wrangle and clean the data using the `filter` and `arrange` functions
- **Today**, we will continue to wrangle and clean the data using the `tidyr` package in `tidyverse` and discuss how data transformations can be performed using `select`, `mutate`, and `transmute` functions, as well as start some data visualization.

# Warm up

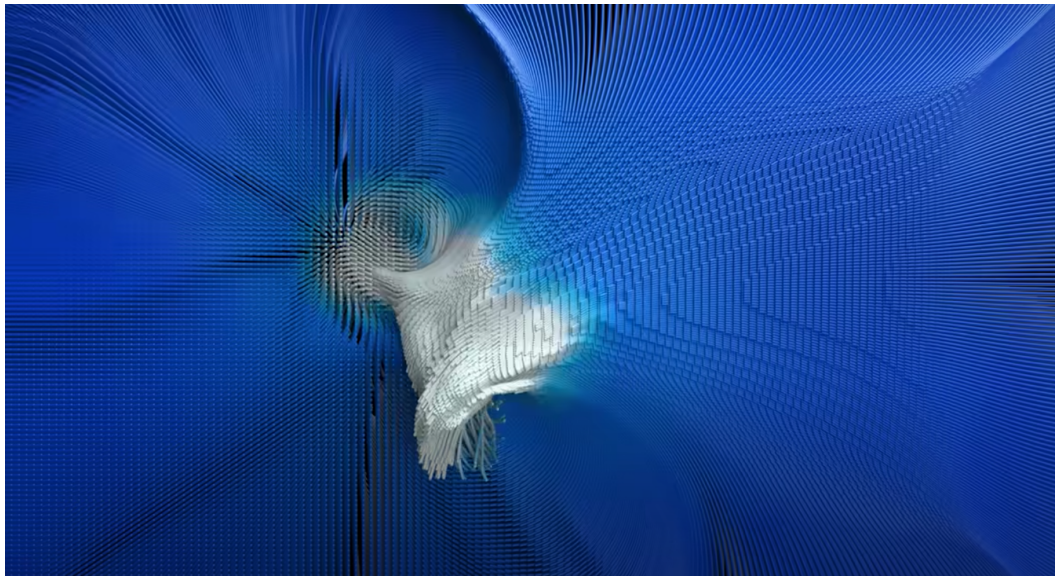


- Can data be **art?** Let's check out this video on how data visualization and machine learning can be used to create a mutual experience:

  ***https://www.youtube.com/watch?v=I-EIVlHvHRM&ab_channel=WIRED***

- We will only watch up to the 2:27 mark but feel free to watch it all later

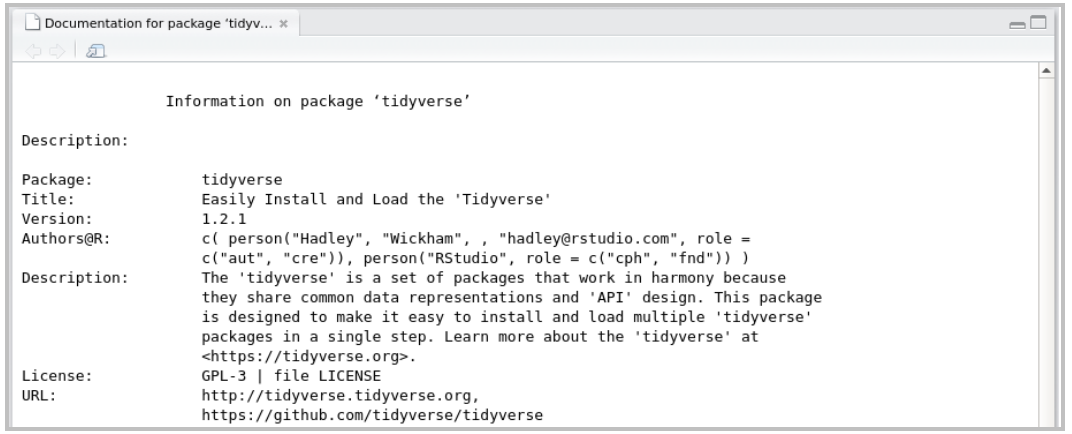
# Chat question



- So, while this is an extreme example, **How/why can artistry in data be useful?**
- Answer in the chat

# Recap: Installing packages

- Before diving deeper and exploring the data cleaning functions in tidyverse, let's do a quick recap on how packages are installed
- Here is an example of how we install and load packages with function **`install.packages()`**
- You can always check the detailed documentation of a package with `help = "package name"`




```
install.packages("tidyverse")    #<- Install
package
library(tidyverse)               #<- Load the
package into the environment.
library(help = "tidyverse")      #<- View package
documentation.
```

# Recap: Installing packages and loading data

- To review the functions within various R packages, we will need to import our dataset
- R comes with several **built-in** data packages. The following is a list of some of the most common datasets:
  - **Titanic**: Survival of passengers on the Titanic
  - **iris**: Edgar Anderson's Iris Data
  - **mtcars**: Motor Trend Car Road Tests
- Today we'll be using one built-in dataset from R called **`nycflights13`** which describes **airline on-time information for all flights departing NYC in 2013**

# Recap: Installing packages and loading data

- Let's now install and load the `nycflights13` package

```
#install.packages("nycflights13")
library(nycflights13)
```

- The `nycflights13` package contains the following five datasets:
  - `flights`: all flights that departed from NYC in 2013
  - `weather`: hourly meteorological data for each airport
  - `planes`: construction information about each plane
  - `airports`: airport names and locations
  - `airlines`: translation between two letter carrier codes and names

# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | |
| Derive new variables from the existing variables using the mutate and transmute commands | |
| Summarize columns using the summarise and group by functions | |
| Understand tidy data and its advantages | |
| Manipulate columns by using the separate and unite functions | |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | |
| Describe and build univariate plots to illustrate patterns in data | |

# Select

- `select` helps you select specific columns within your dataframe

```
# Check for detailed documentation
?dplyr::select

# Use cases for `select` function
select(df,            #<- dataframe
       select_cond1, #<- selection rule(s)
       ...)
```

- We often use this function with pipes(%>%) which we will cover later in the course
- The selection criteria can be written in multiple ways, as shown in the next couple of slides

**Usage**

```
select(.data, ...)
```

**Arguments**

| | |
|---|---|
| `.data` | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |
| `...` | <tidy-select> One or more unquoted expressions separated by commas. Variable names can be used as if they were positions in the data frame, so expressions like `x:y` can be used to select a range of variables. |

# Select a subset

- Simply specify the column name(s)

```
# Select columns from `flights` dataframe.
library(tidyverse)
select(flights, #<- specify the dataframe
       year,    #<- specify the 1st column
       month,   #<- specify the 2nd column
       day)     #<- specify the 3rd column
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# … with 336,766 more rows
```

- You can also specify a range of columns with the range operator (i.e. `:`)

```
# Select columns from `flights` dataframe
select(flights,  #<- specify the dataframe
       year:day) #<- specify the range of
columns
```

```
# A tibble: 336,776 x 3
    year month   day
   <int> <int> <int>
 1  2013     1     1
 2  2013     1     1
 3  2013     1     1
 4  2013     1     1
 5  2013     1     1
 6  2013     1     1
 7  2013     1     1
 8  2013     1     1
 9  2013     1     1
10  2013     1     1
# … with 336,766 more rows
```

# Select by excluding

- Finally, you can select by excluding certain columns using the exclusion operator (i.e. –)

```
# Select multiple columns from `flights` dataframe by providing which columns to exclude in selection
select(flights,     #<- specify the dataframe
       -(year:day)) #<- specify the range of columns to exclude
```

```
# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
      <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
 1      517            515         2      830            819        11 UA
 2      533            529         4      850            830        20 UA
 3      542            540         2      923            850        33 AA
 4      544            545        -1     1004           1022       -18 B6
 5      554            600        -6      812            837       -25 DL
 6      554            558        -4      740            728        12 UA
 7      555            600        -5      913            854        19 B6
 8      557            600        -3      709            723       -14 EV
 9      557            600        -3      838            846        -8 B6
10      558            600        -2      753            745         8 AA
# … with 336,766 more rows, and 9 more variables: flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

# Select - helper functions

- Helpers are multiple functions you can use to select variables based on their names
- They act like regular expressions, but in a more simplified manner
- Here are some of the more commonly used helper functions:

| Helper Function | Use Case |
|---|---|
| `starts_with("abc")` | matches names that begin with “abc” |
| `ends_with("xyz")` | matches names that end with “xyz” |
| `contains("ijk")` | matches names that contain “ijk” |
| `num_range("x", 1:3)` | matches “x1”, “x2” and “x3” |

- To select columns whose names start with 'arr':

```
select(flights, starts_with("arr"))
```

# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | |
| Summarize columns using the summarise and group by functions | |
| Understand tidy data and its advantages | |
| Manipulate columns by using the separate and unite functions | |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | |
| Describe and build univariate plots to illustrate patterns in data | |

# Mutate

- `mutate` is an essential function of dplyr
- It allows us to **create** new variables using the current data and **append** these variables to the existing dataframe

```
?dplyr::mutate

mutate(df,        # <- dataframe
       new_col1, # <- rule(s) for the new column
       ...)
```

- `mutate` always adds columns to the end of the dataset, so make sure you are able to see the last columns

mutate {dplyr} R Documentation

## Create, modify, and delete columns

### Description

`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`.

### Usage

```
mutate(.data, ...)

## S3 method for class 'data.frame'
mutate(
  .data,
  ...,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)

transmute(.data, ...)
```

### Arguments

| | |
|---|---|
| `.data` | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |

# Mutate - create the dataset

- Create the dataset using `select`

```
# Let's select columns of `flights` dataframe and save them as `flights_sml`.
flights_sml = select(flights,            #<- specify data fra,e
                     year:day,           #<- specify range of columns to include
                     ends_with("delay"), #<- find all columns that end with `delay`
                     distance,           #<- select `distance` column
                     air_time)           #<- select `air_time` column
flights_sml
```

```
# A tibble: 336,776 x 7
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1         2        11     1400      227
 2  2013     1     1         4        20     1416      227
 3  2013     1     1         2        33     1089      160
 4  2013     1     1        -1       -18     1576      183
 5  2013     1     1        -6       -25      762      116
 6  2013     1     1        -4        12      719      150
 7  2013     1     1        -5        19     1065      158
 8  2013     1     1        -3       -14      229       53
 9  2013     1     1        -3        -8      944      140
10  2013     1     1        -2         8      733      138
# … with 336,766 more rows
```

# Mutate - arguments

- The first argument is the **dataframe**
- The following arguments are the **columns we would like to add to the dataframe**

```
# Add two columns `gain` and `speed` to `flights_sml`.
mutate(flights_sml,                          #<- specify the dataframe
       gain = arr_delay - dep_delay,         #<- create `gain` column by subtracting departure delay
                                             #   from arrival delay
       speed = distance / air_time * 60)     #<- create `speed` from distance and air time columns
```

```
# A tibble: 336,776 x 9
    year month   day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9  370.
 2  2013     1     1         4        20     1416      227    16  374.
 3  2013     1     1         2        33     1089      160    31  408.
 4  2013     1     1        -1       -18     1576      183   -17  517.
 5  2013     1     1        -6       -25      762      116   -19  394.
 6  2013     1     1        -4        12      719      150    16  288.
 7  2013     1     1        -5        19     1065      158    24  404.
 8  2013     1     1        -3       -14      229       53   -11  259.
 9  2013     1     1        -3        -8      944      140    -5  405.
10  2013     1     1        -2         8      733      138    10  319.
# … with 336,766 more rows
```

# Transmute

- `transmute` is a function that does the same thing as `mutate` **except it will only keep the new columns**

```
transmute(df,        # <- dataframe
          new_col1, # <- rule(s) for new column
          ...)
```

- The first argument is the **dataframe**
- The following arguments are the **columns that will be included in your new dataframe**
- *Note: you are isolating **only** these new columns*

mutate {dplyr} R Documentation

## Create, modify, and delete columns

### Description

`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`.

### Usage

```
mutate(.data, ...)

## S3 method for class 'data.frame'
mutate(
  .data,
  ...,
  .keep = c("all", "used", "unused", "none"),
  .before = NULL,
  .after = NULL
)

transmute(.data, ...)
```

### Arguments

| | |
|---|---|
| `.data` | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |

# Transmute example

- With the same arguments as in the `mutate` example, we can see that the `transmute` function returns only new columns

```
# Add two columns `gain` and `speed` to `flights_sml`.
transmute(flights_sml,                          #<- specify the dataframe
          gain = arr_delay - dep_delay,         #<- create `gain` column by subtracting departure delay
                                                #   from arrival delay
          speed = distance / air_time * 60) #<- create `speed` from distance and air time columns
```

```
# A tibble: 336,776 x 2
    gain speed
   <dbl> <dbl>
 1     9  370.
 2    16  374.
 3    31  408.
 4   -17  517.
 5   -19  394.
 6    16  288.
 7    24  404.
 8   -11  259.
 9    -5  405.
10    10  319.
# … with 336,766 more rows
```

# Mutate and transmute - useful functions

- When creating new variables with `mutate/transmute`, there are many helpful widgets and functions that can assist in creating interesting features:

| Useful Functions | Explanation |
|---|---|
| `+, -, *, /, ^` | all mathematical operators can be used on variables |
| `log, log2, log10` | logarithmic functions for variable transformation can be used |
| `%/%` and `%%` | modulus and remainder are useful when converting time |
| `lag(x)` and `lead(x)` | lag and lead allow reference to leading or lagging values - useful for detecting changes in values. |
| `cumsum(x),cummean(x),` `cummax(x),cumprod(x)` | cumulative, running functions, mins, max, prod, mean, etc. |

# Mutate and transmute - useful functions (cont-d)

- **Ranking functions** are very helpful in data manipulation
- There are several within the `dplyr` package such as `row_number()`, `ntile()` and `dense_rank()`

```
# Check for detailed documentation
?dplyr::ranking

rank_function(x) # <- one of rank functions with
                 #    a vector of values to rank
```

ranking {dplyr} R Documentation

## Windowed rank functions.

### Description

Six variations on ranking functions, mimicking the ranking functions described in SQL2003. They are currently implemented using the built in `rank` function, and are provided mainly as a convenience when converting between R and SQL. All ranking functions map smallest inputs to smallest outputs. Use `desc()` to reverse the direction.

### Usage

```
row_number(x)

ntile(x = row_number(), n)

min_rank(x)

dense_rank(x)

percent_rank(x)

cume_dist(x)
```

### Arguments

| | |
|---|---|
| `x` | a vector of values to rank. Missing values are left as is. If you want to treat them as the smallest or largest values, replace with Inf or -Inf before ranking. |
| `n` | number of groups to split up into. |

# Knowledge Check 1

# Exercise 1

# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | ✔ |
| Summarize columns using the summarise and group by functions | |
| Understand tidy data and its advantages | |
| Manipulate columns by using the separate and unite functions | |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | |
| Describe and build univariate plots to illustrate patterns in data | |

# Directory settings and loading packages

- In order to maximize the efficiency of your workflow, you may want to encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `skillsoft-2021` folder

```
# Set `main_dir` to the location of your `skillsoft-2021` folder (for Mac/Linux).
main_dir = "~/Desktop/skillsoft-2021"
# Set `main_dir` to the location of your `skillsoft-2021` folder (for Windows).
main_dir = "C:/Users/[username]/Desktop/skillsoft-2021"

# Make `data_dir` from the `main_dir` and remainder of the path to data directory.
data_dir = paste0(main_dir, "/data")
# Make `plots_dir` from the `main_dir` and remainder of the path to plots directory.
plot_dir = paste0(main_dir, "/plots")

# Set directory to data_dir.
setwd(data_dir)
```

```
library(tidyverse)
library(nycflights13)
```

- Today we'll continue working with the **`nycflights13`** dataset, which describes **airline on-time information for all flights departing NYC in 2013**

# Installing packages and loading data

- `.RData` is a specific format designed for storing a complete R workspace, or selected objects from a workspace, in a form that can be easily loaded back into R
- RData files are organized as a sequence of objects while each object has a type
- We will get back to it with a more detailed introduction later in this module

```
setwd(data_dir)
load("tidyr_tables.RData")
```

# Summarise and group_by

- `summarise` collapses a dataframe down to a single row
- By itself, `summarise` is not very helpful
- We will often use it with `group_by`
- **Note:** You can also use `summarize`

```
# Check for detailed documentation
?dplyr::summarise

# Use cases for `summarise` function
summarise(df,                #<- dataframe
          summary_function1, #<- summary rule(s)
                             #   for new column
          ...)
```

summarise {dplyr} R Documentation

## Summarise each group to fewer rows

### Description

`summarise()` 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 summarising all observations 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.

`summarise()` and `summarize()` are synonyms.

### Usage

```
summarise(.data, ..., .groups = NULL)

summarize(.data, ..., .groups = NULL)
```

### Arguments

| | |
|---|---|
| `.data` | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |

# Summarise and group_by

- Grouping doesn't change how data looks apart from listing how it's grouped
- It will change how it acts with the other dplyr verbs
- To remove grouping, use `ungroup`

```
# Check for detailed documentation
?dplyr::group_by

# Use cases for `group_by` function
group_by(df,          #<- dataframe
         variable1, #<- 1st variable to group
by
         variable2, #<- 2nd variable to group
by
         ...)
```

group_by {dplyr} R Documentation

## Group by one or more variables

### Description

Most data operations are done on groups defined by variables. `group_by()` takes an existing tbl and converts it into a grouped tbl where operations are performed "by group". `ungroup()` removes grouping.

### Usage

```
group_by(.data, ..., .add = FALSE, .drop = group_by_drop_default(.data))

ungroup(x, ...)
```

### Arguments

| | |
|---|---|
| `.data` | A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See *Methods*, below, for more details. |
| `...` | In `group_by()`, variables or computations to group by. In `ungroup()`, variables to remove from the grouping. |
| `.add` | When `FALSE`, the default, `group_by()` will override existing groups. To add to the existing groups, use `.add = TRUE`.<br>This argument was previously called `add`, but that prevented creating a new grouping variable called `add`, and conflicts with our naming conventions. |
| `.drop` | When `.drop = TRUE`, empty groups are dropped. See `group_by_drop_default()` for what the default value is for this argument. |
| `x` | A `tbl()` |

# Summarise and group_by alone

```
# Produce a summary
summarise(flights, delay =
mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 1 x 1
  delay
  <dbl>
1  12.6
```

```
# Create `by_day` by grouping `flights` by year, month, and day.
by_day = group_by(flights, year, month, day)
by_day
```

```
# A tibble: 336,776 x 19
# Groups:   year, month, day [365]
    year month   day dep_time sched_dep_time dep_delay arr_time
sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
<int>
 1  2013     1     1      517            515         2      830
819
 2  2013     1     1      533            529         4      850
830
 3  2013     1     1      542            540         2      923
850
 4  2013     1     1      544            545        -1     1004
1022
 5  2013     1     1      554            600        -6      812
837
 6  2013     1     1      554            558        -4      740
728
 7  2013     1     1      555            600        -5      913
854
 8  2013     1     1      557            600        -3      709
723
 9  2013     1     1      557            600        -3      838
846
10  2013     1     1      558            600        -2      753
745
# … with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour
<dttm>
```

# Summarise and group_by together

```
# Now use grouped `by_day` data and summarise it to see the average delay by year, month and day.
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# … with 355 more rows
```

- **`summarise`** and **`group_by`** are two of the most widely used functions within dplyr!

# Dplyr and the pipe: without it

- Now we get to the best part: connecting it all
- Let's say we want to do these three things:
  - **Group** flights by destination
  - **Summarize** to compute distance, average delay, and number of flights
  - **Filter** to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport
- We might think we have to write out a `dplyr` function for each, save each as a variable, and then continue to perform the next function, which should look something like this:

```
# Step 1: Create a new grouped data
frame `by_dest`.
by_dest = group_by(flights, dest)

# Step 2: Create a summary of `by_dest` and save as `delay`.
delay = summarise(by_dest,
                  count = n(),
                  dist = mean(distance, na.rm = TRUE),
                  delay = mean(arr_delay, na.rm = TRUE))

# Step 3: Filter `delay` by their count and destination.
delay = filter(delay, count > 20, dest != "HNL")
```

# Dplyr and the pipe: a better way

- Sure, that works, but can we do it cleaner? Faster? **YES!**
- We can use the **pipe operator** (i.e. %>%) and do it all in a single step without creating extra variables

```
delays = flights %>%                                          #<- take flights data
  group_by(dest) %>%                                          #<- group it by destination
  summarise(count = n(),                                      #<- then summarize by creating count variable
            dist = mean(distance, na.rm = TRUE),              #<- and computing mean distance
            delay = mean(arr_delay, na.rm = TRUE)) %>%        #<- and mean arrival delay
  filter(count > 20, dest != "HNL")                           #<- then filter it
delays
```

```
# A tibble: 96 x 4
   dest  count  dist delay
   <chr> <int> <dbl> <dbl>
 1 ABQ     254 1826   4.38
 2 ACK     265  199   4.85
 3 ALB     439  143  14.4
 4 ATL   17215  757. 11.3
 5 AUS    2439 1514.  6.02
 6 AVL     275  584.  8.00
 7 BDL     443  116   7.05
 8 BGR     375  378   8.03
 9 BHM     297  866. 16.9
10 BNA    6333  758. 11.8
# … with 86 more rows
```

# Summarise and handling NAs

## We do NOT address NAs

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# … with 355 more rows
```

- If we do not address NAs, the aggregation functions will return NAs for each item if there is just one NA in the input

## We address NAs

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay,
                        na.rm = TRUE))
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# … with 355 more rows
```

- **Moral of the story:** remember to address NAs when using summarise!

# A few more useful summary functions

- Apart from `mean()`, there are many other summary functions that describe data from various aspects:

| Summary Functions | Explanation |
|---|---|
| `n()` | will count the number of entries that come from a summarise |
| `min(x),quantile(x, 0.25),max(x)` | measures of rank and distribution can be used |
| `first(x),nth(x, 2),last(x)` | measures of position and order |
| `n_distinct` | will count the number of distinct values |

# Summarise n to count

- `n` will count the number of entries that come from a `summarise` function

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE),
            n = n()) #<- add a column with summary counts
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day  mean     n
   <int> <int> <int> <dbl> <int>
 1  2013     1     1 11.5    842
 2  2013     1     2 13.9    943
 3  2013     1     3 11.0    914
 4  2013     1     4  8.95   915
 5  2013     1     5  5.73   720
 6  2013     1     6  7.15   832
 7  2013     1     7  5.42   933
 8  2013     1     8  2.55   899
 9  2013     1     9  2.28   902
10  2013     1    10  2.84   932
# … with 355 more rows
```

# Summarise not needed to count

- `count` is a simple count function that does not require the summary function

```
flights %>%
  count(day) #<- count number of instances of entry in `day` column
```

```
# A tibble: 31 x 2
     day     n
 * <int> <int>
 1     1 11036
 2     2 10808
 3     3 11211
 4     4 11059
 5     5 10858
 6     6 11059
 7     7 10985
 8     8 11271
 9     9 10857
10    10 11227
# … with 21 more rows
```

# Summarise rank

- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`

```
flights %>%
  group_by(year, month) %>%
  summarise(first = min(dep_time, na.rm = TRUE),
            last = max(dep_time, na.rm = TRUE))
```

```
# A tibble: 12 x 4
# Groups:   year [1]
    year month first  last
   <int> <int> <int> <int>
 1  2013     1     1  2359
 2  2013     2     1  2400
 3  2013     3     1  2400
 4  2013     4     1  2400
 5  2013     5     1  2400
 6  2013     6     1  2400
 7  2013     7     1  2400
 8  2013     8     1  2400
 9  2013     9     2  2400
10  2013    10     6  2400
11  2013    11     1  2400
12  2013    12     1  2400
```

# Summarise position

```
# 1. Build a subset of all flights that were not canceled.
not_cancelled = flights %>%
  filter(!is.na(dep_time))  #<- filter flights where `dep_time` was not `NA`

# 2. Group and summarize all flights that were not canceled to get desired results.
not_cancelled  %>%
  group_by(year, month, day) %>%   #<- group the not canceled flights
  summarise(first = min(dep_time), #<- then summarize them by calculating the first
            last = max(dep_time))  #<- and last flights in the `dep_time` in each group
```

```
# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day first  last
   <int> <int> <int> <int> <int>
 1  2013     1     1   517  2356
 2  2013     1     2    42  2354
 3  2013     1     3    32  2349
 4  2013     1     4    25  2358
 5  2013     1     5    14  2357
 6  2013     1     6    16  2355
 7  2013     1     7    49  2359
 8  2013     1     8   454  2351
 9  2013     1     9     2  2252
10  2013     1    10     3  2320
# … with 355 more rows
```

# Summarise distinct values

- `n_distinct(x)` will count the number of distinct values

```
# Number of flights that take off, by day.
not_cancelled  %>%
  group_by(year, month, day) %>%
  summarise(flights_that_take_off = n_distinct(dep_time)) #<- calculate distinct departure times
```

```
# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day flights_that_take_off
   <int> <int> <int>                 <int>
 1  2013     1     1                   552
 2  2013     1     2                   583
 3  2013     1     3                   589
 4  2013     1     4                   589
 5  2013     1     5                   495
 6  2013     1     6                   564
 7  2013     1     7                   572
 8  2013     1     8                   573
 9  2013     1     9                   580
10  2013     1    10                   572
# … with 355 more rows
```

# Remember to ungroup before you regroup

```
# Take the same `not_canceled` data, but now group by month instead of by day.
not_cancelled %>%                                    #<- set dataframe
  ungroup() %>%                                      #<- first ungroup it
  group_by(year, month) %>%                          #<- then group by year and month
  summarise(flights_by_year = n_distinct(dep_time))#<- then do the rest ...
```

```
# A tibble: 12 x 3
# Groups:   year [1]
    year month flights_by_year
   <int> <int>           <int>
 1  2013     1            1165
 2  2013     2            1171
 3  2013     3            1199
 4  2013     4            1216
 5  2013     5            1186
 6  2013     6            1220
 7  2013     7            1242
 8  2013     8            1204
 9  2013     9            1156
10  2013    10            1139
11  2013    11            1135
12  2013    12            1191
```

# Knowledge Check 2

# Exercise 2

# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | ✔ |
| Summarize columns using the summarise and group by functions | ✔ |
| Understand tidy data and its advantages | |
| Manipulate columns by using the separate and unite functions | |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | |
| Describe and build univariate plots to illustrate patterns in data | |

# Data wrangling

- Data transformation is where you get the dataset ready for wrangling
- We want all the variables and values, all the new columns to be created, and all the NAs taken care of before ensuring it is in `tidy` form
- `tidyr`, the package within `tidyverse`, allows us to get our data into a tidy format
- We will use the `.Rdata` file loaded at the beginning of this lesson to demonstrate

- For further reading and understanding of tidy data and where it originated, check out this ***paper*** by Hadley Wickham, Chief Scientist at RStudio

# Would analysis be easy with these datasets?

- Here is a list of objects from our RData file

```
key_value_country
```

```
# A tibble: 12 x 4
   country      year key             value
   <fct>       <int> <fct>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population    19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population    20595360
 5 Brazil       1999 cases            37737
 6 Brazil       1999 population   172006362
 7 Brazil       2000 cases            80488
 8 Brazil       2000 population   174504898
 9 China        1999 cases           212258
10 China        1999 population  1272915272
11 China        2000 cases           213766
12 China        2000 population  1280428583
```

```
year_country
```

```
# A tibble: 3 x 3
  country     `1999` `2000`
  <fct>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

```
rate_country
```

```
# A tibble: 6 x 3
  country      year rate
  <fct>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

# What makes data 'tidy'?

- These three interrelated rules make a dataset tidy:
  - Each **variable** must have its own **column**
  - Each **observation** must have its own **row**
  - Each **value** must have its own **cell**
- `tidy_country` is the only table that follows all three rules

```
tidy_country
```

```
# A tibble: 6 x 4
  country      year  cases population
  <fct>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

# What are the advantages of tidy data?

- Storing data in a **consistent** way:
  - It's easier to learn the tools that work with it because of the underlying uniformity
- Making use of R's **internal vectorization**:
  - Most built-in R functions work with vectors of values

# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | ✔ |
| Summarize columns using the summarise and group by functions | ✔ |
| Understand tidy data and its advantages | ✔ |
| Manipulate columns by using the separate and unite functions | |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | |
| Describe and build univariate plots to illustrate patterns in data | |

# Separating and uniting

- But how would we adjust a single variable?
- What would we use for a dataframe like `rate_country`?

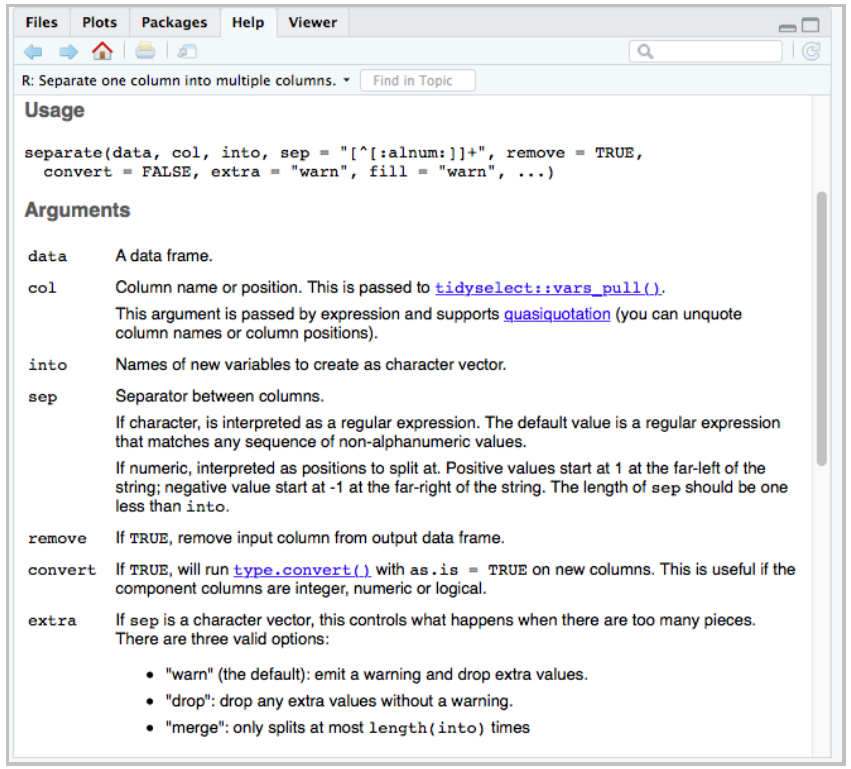
```
rate_country
```

```
# A tibble: 6 x 3
  country      year rate
  <fct>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

- What do we do with the `rate` column?
  - We can use the function **`separate`**
- The complement of `separate` is **`unite`**
- We will learn how to use `unite` as well

# Separate

- `separate` separates a single character column into multiple columns and takes two arguments:
  - The first argument is the **dataframe**
  - Next we pipe it to `separate`
  - The first parameter is the **column** to be separated
  - The second parameter defines **how we want to separate the variable**, using `into = c("var_1", "var_2")`




```
?tidyr::separate

separate(df,    #<- dataframe
         col,   #<- name of column to separate
         into)  #<- name of new variables to
                #   create as a character vector
```

# Separate

```
# Using `rate_country` separate its `rate` column into two.
rate_country %>%                    #<- set dataframe and pass it to next function with pipe
  separate(rate,                    #<- separate `rate`
           into = c("cases",        #<- into column `cases`, and
                    "population")) #<-      column `population`
```

```
# A tibble: 6 x 4
  country      year cases  population
  <fct>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

# Separate

- By default, `separate` will separate on any non alpha-numeric character
- However, you can also specify the character by which to separate

```
# Using `rate_country` separate its `rate` column into two.
rate_country %>%
  separate(rate,
           into = c("cases",
                    "population"),
           sep = "/")              #<- set the separating character to `/`
```

```
# A tibble: 6 x 4
  country      year cases  population
  <fct>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

# Separate: sep set to index

- You can use the `sep` parameter to separate the year column on the **character index** into `century` and `year`

```
# Using `rate_country` separate its `year` column into two.
rate_country %>%
  separate(year,              #<- separate `year`
           into= c("century", #<- into two columns: `century`, and
                   "year"),   #<-                   `year`
           sep = 2)           #<- set the separator at index = 2
```

```
# A tibble: 6 x 4
  country     century year  rate
  <fct>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

# Separate: data type conversion

- When we use `separate`, the data type of the original column will be preserved

```
# The new columns
# are now also characters.
rate_country %>%
  separate(rate, into = c("cases",
"population"))
```

```
# A tibble: 6 x 4
  country      year  cases  population
  <fct>        <int> <chr>  <chr>
1 Afghanistan  1999  745    19987071
2 Afghanistan  2000  2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999  212258 1272915272
6 China        2000  213766 1280428583
```

- However, we can tell separate to convert to what it thinks the data types of new columns should be

```
rate_country %>%
  separate(rate, into = c("cases", "population"), convert =
TRUE)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <fct>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

# Unite

```
?tidyr::unite

unite(df,  #<- dataframe
      col, #<- name of column to unite
      sep) #<- separator to use
```

- `unite` combines multiple character columns into a single column
- `unite` is the inverse of separate

unite {tidyr} R Documentation

## Unite multiple columns into one by pasting strings together

### Description

Convenience function to paste together multiple columns into one.

### Usage

```
unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)
```

### Arguments

| | |
|---|---|
| `data` | A data frame. |
| `col` | The name of the new column, as a string or symbol.<br>This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with `rlang::ensym()` (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility). |
| `...` | <`tidy-select`> Columns to unite |
| `sep` | Separator to use between values. |
| `remove` | If `TRUE`, remove input columns from output data frame. |
| `na.rm` | If `TRUE`, missing values will be remove prior to uniting each value. |

# Unite example

- We will use the separated-on-year example of `rate_country` to show `unite`

```
# Let's separate the `rate_country`'s `year`
column into `century` and `year` first.
ex_table = rate_country %>%
  separate(year, into = c("century", "year"),
sep = 2)
ex_table
```

```
# A tibble: 6 x 4
  country     century year  rate
  <fct>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

```
# Now we use `unite` to combine the two new
columns back into one.
# By default, unite will combine columns using
`_` so we can use `sep` to specify that we
# do not want anything between the two columns
when combined into one cell.
ex_table %>%      #<- specify the dataframe to
pipe into `unite`
  unite(time,     #<- set the column `time` for
combined values
        century,  #<- 1st column to unite
        year,     #<- 2nd column to unite
        sep = "") #<- set the separator to an
empty string
```

```
# A tibble: 6 x 3
  country     time  rate
  <fct>       <chr> <chr>
1 Afghanistan 1999  745/19987071
2 Afghanistan 2000  2666/20595360
3 Brazil      1999  37737/172006362
4 Brazil      2000  80488/174504898
5 China       1999  212258/1272915272
6 China       2000  213766/1280428583
```

# Module completion checklist

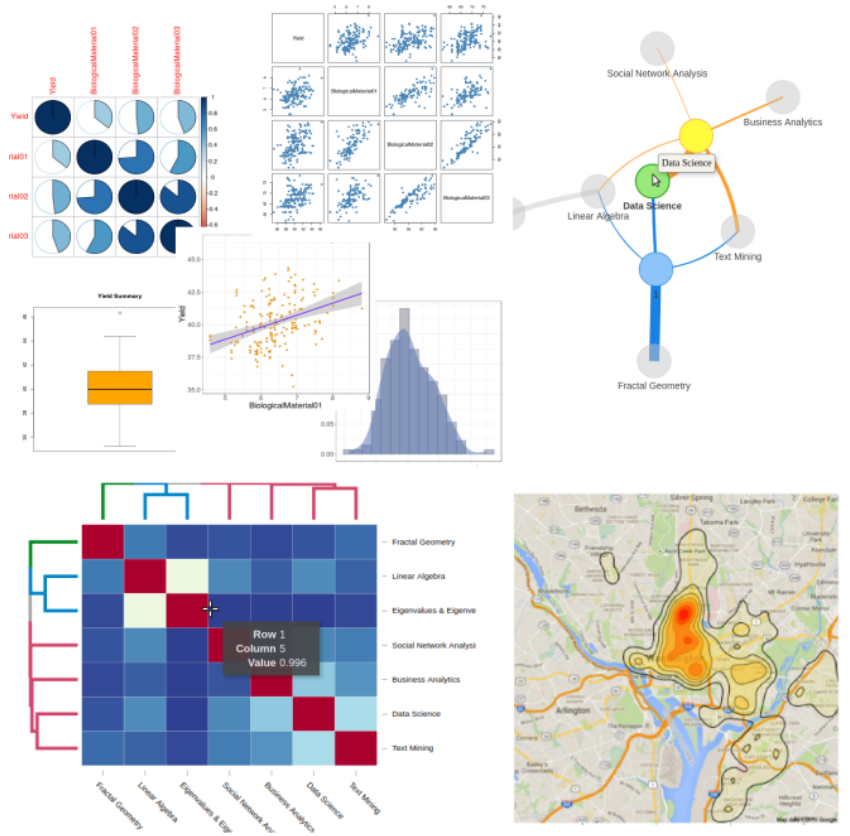
| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | ✔ |
| Summarize columns using the summarise and group by functions | ✔ |
| Understand tidy data and its advantages | ✔ |
| Manipulate columns by using the separate and unite functions | ✔ |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | |
| Describe and build univariate plots to illustrate patterns in data | |

# Why visualize?

Why do we need to build visualizations?

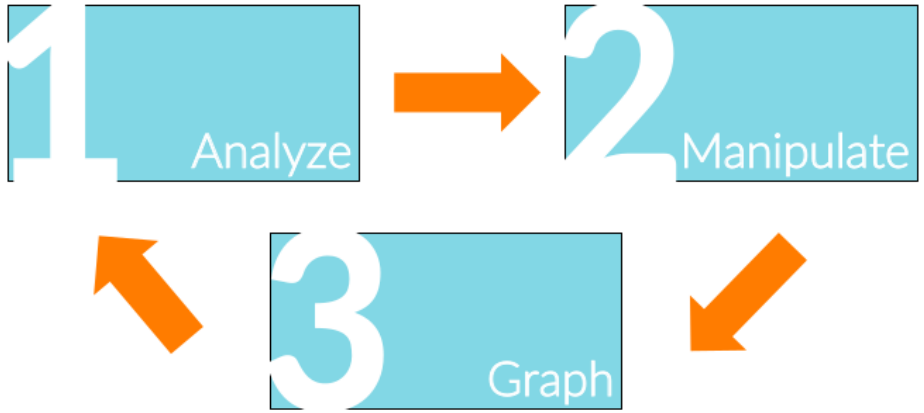
- To provide valuable insights that are interpretable and relevant
- To give a visual or graphical representation of data / concepts
- To communicate ideas
- To provide an accessible way to see and understand trends, outliers, and patterns in data
- To confirm a hypothesis about the data

The images on the right show some common types of charts and graphs used in data science



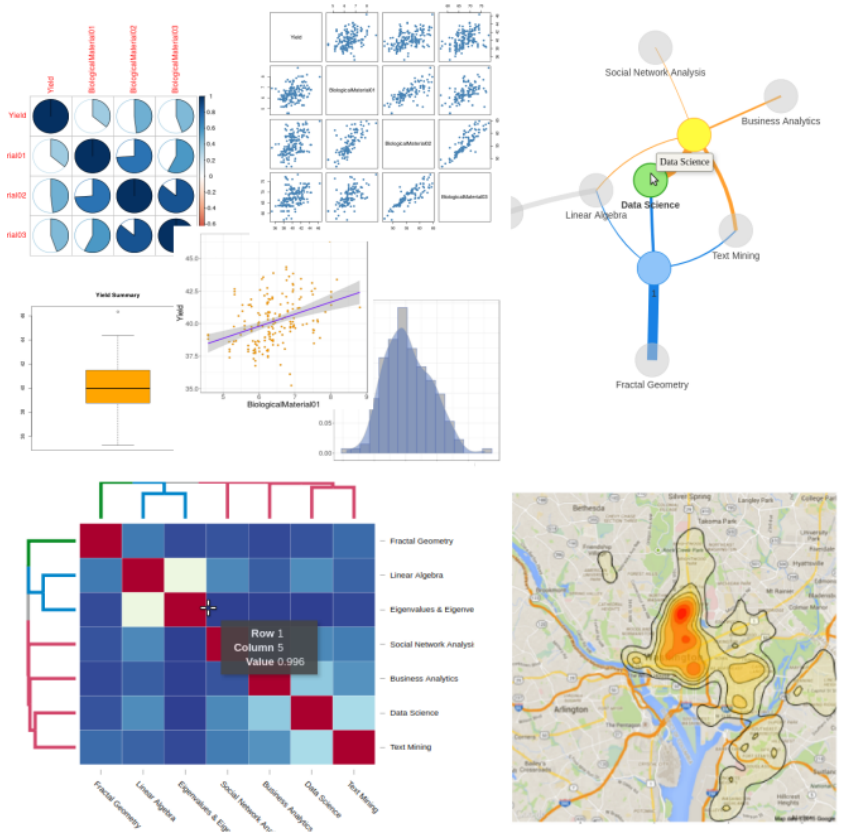
# Exploratory Data Analysis (EDA)

- R is a powerful tool for EDA because the graphics tie in with the functions used to analyze data. You can create graphs **without** breaking your train of thought as you explore your data
- As the image on the right indicates, visualization is an **iterative process** that consists of the following steps:
  - Analyze data
  - Manipulate data
  - Graph the results
  - Repeat



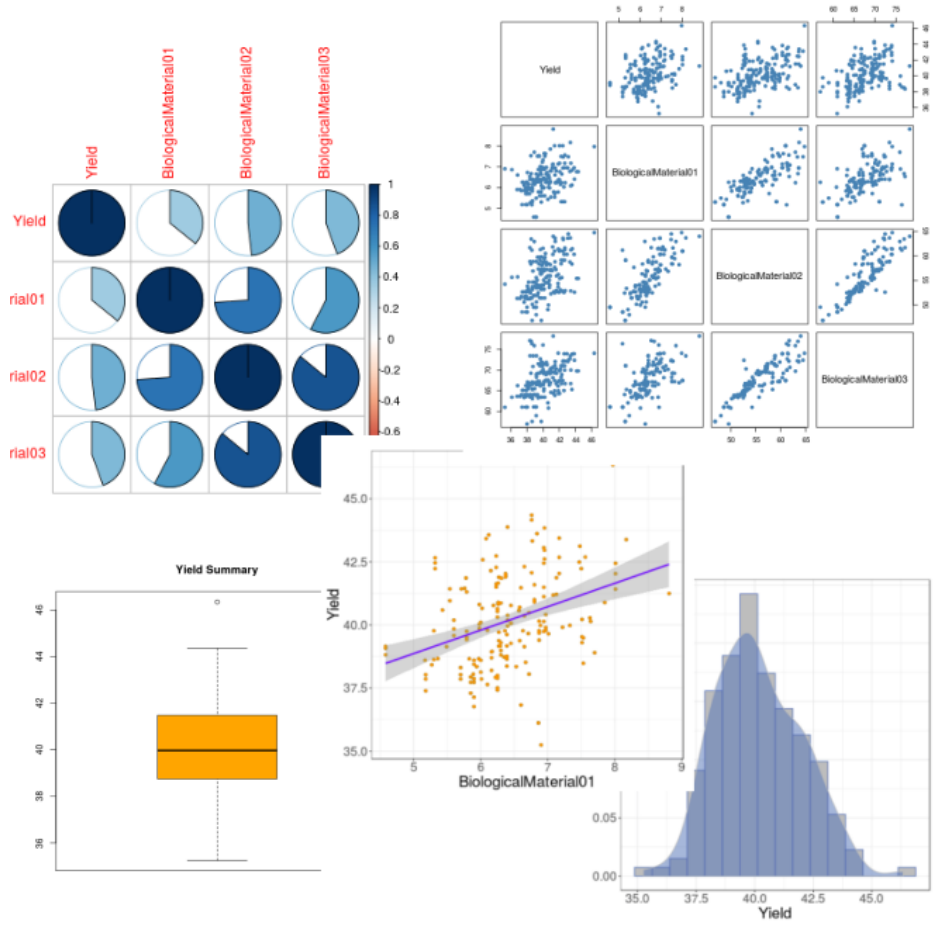
# Visualizing data in R

- R comes with multiple types of visualizations:
  - Basic plots & composite graphs
  - Maps
  - Dynamic visualizations
  - Interactive charts & dashboards
  - 3D graphics
- You can also save and share your work in a variety of file formats, and with adjustable image quality (`SVG`, `PNG`, `JPEG`, `PDF`, etc.)
- *Use **this link** to see a list of help pages, vignettes, and code demos.*



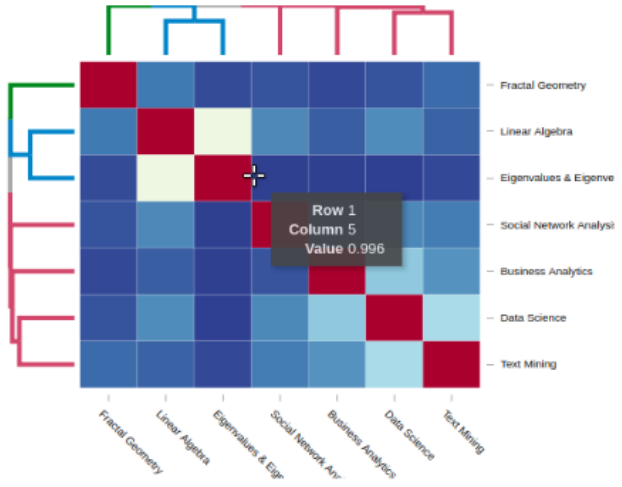
# Static visualizations in R

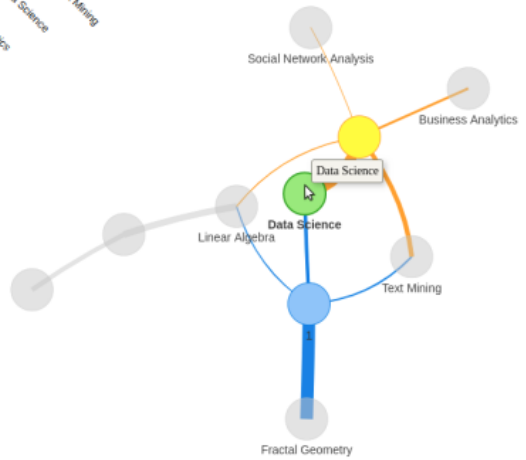
- **Static plots**
  - Only **display** data, **without letting the user interact** with the visualization
  - Are available through base R and multitudes of packages (e.g., `ggplot2`, `corrplot`)
  - Are high quality (R visualizations are made in scalable vector graphics format, or `SVG`) and can be saved as `SVG`, `PNG`, `JPEG`, `BMP`, or `PDF`
  - Are the best way to display patterns in data for **printed publications**



# Interactive visualizations in R

- **Interactive plots**
  - **Display** data and also **let the user interact** with the visualization by clicking, dragging, zooming, etc.
  - Are available through many packages (e.g., `highcharter`, `plotly`, `htmlwidgets`)
  - Are high quality (graphical elements are in SVG format), render as `HTML` pages, and can be saved as `HTML` documents and opened through browser
  - Are the best way to display patterns in data for **web publications**, on **websites**, in **web applications**



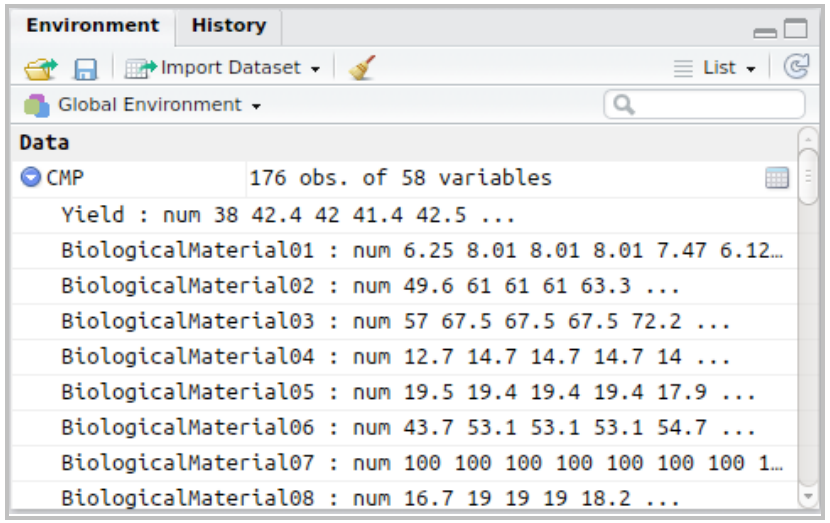
# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | ✔ |
| Summarize columns using the summarise and group by functions | ✔ |
| Understand tidy data and its advantages | ✔ |
| Manipulate columns by using the separate and unite functions | ✔ |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | ✔ |
| Describe and build univariate plots to illustrate patterns in data | |

# Loading the CMP dataset for EDA

- We will be using the **Chemical Manufacturing Process** (CMP) dataset to create various static visualizations
- Let's load the dataset from our `data_dir` into R's environment

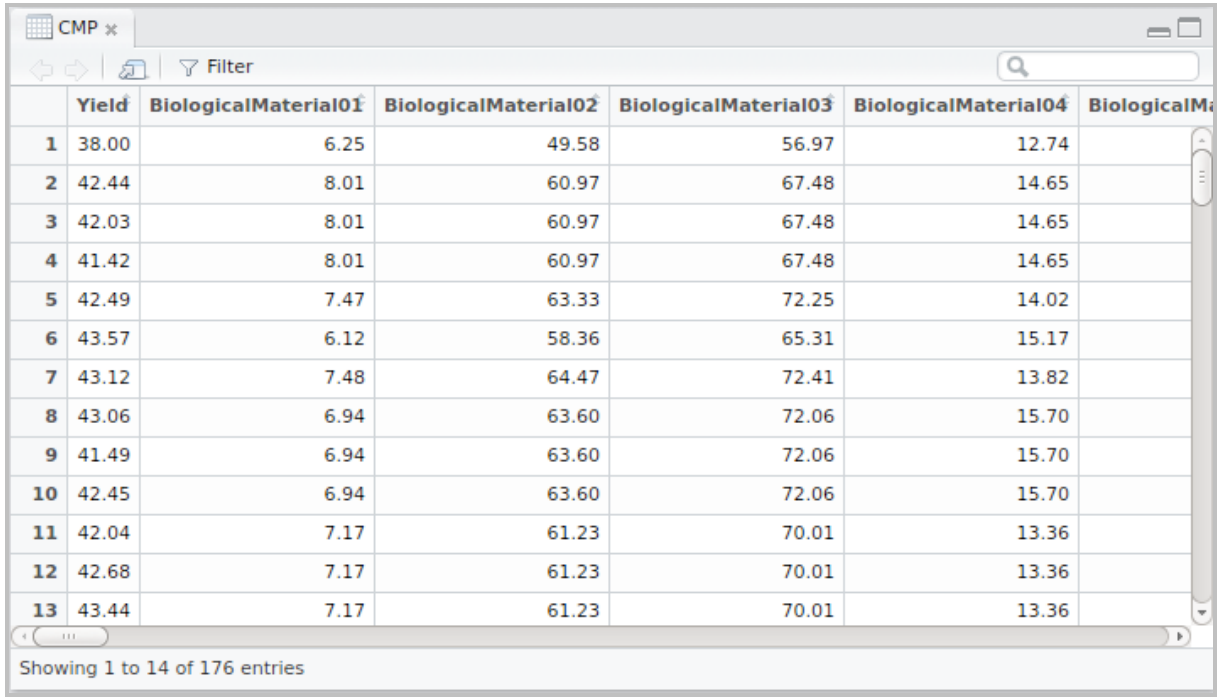
```
# Set working directory to where we store data.
setwd(data_dir)

# Read CSV file called "ChemicalManufacturingProcess.csv"
CMP = read.csv("ChemicalManufacturingProcess.csv",
               header = TRUE,
               stringsAsFactors = FALSE)
```



# About the CMP dataset

- The dataset consists of 176 observations and 58 variables

```
# View CMP dataset in the tabular data explorer.
View(CMP)
```



CMP

Filter

| | Yield | BiologicalMaterial01 | BiologicalMaterial02 | BiologicalMaterial03 | BiologicalMaterial04 | BiologicalMa |
|---|---|---|---|---|---|---|
| 1 | 38.00 | 6.25 | 49.58 | 56.97 | 12.74 | |
| 2 | 42.44 | 8.01 | 60.97 | 67.48 | 14.65 | |
| 3 | 42.03 | 8.01 | 60.97 | 67.48 | 14.65 | |
| 4 | 41.42 | 8.01 | 60.97 | 67.48 | 14.65 | |
| 5 | 42.49 | 7.47 | 63.33 | 72.25 | 14.02 | |
| 6 | 43.57 | 6.12 | 58.36 | 65.31 | 15.17 | |
| 7 | 43.12 | 7.48 | 64.47 | 72.41 | 13.82 | |
| 8 | 43.06 | 6.94 | 63.60 | 72.06 | 15.70 | |
| 9 | 41.49 | 6.94 | 63.60 | 72.06 | 15.70 | |
| 10 | 42.45 | 6.94 | 63.60 | 72.06 | 15.70 | |
| 11 | 42.04 | 7.17 | 61.23 | 70.01 | 13.36 | |
| 12 | 42.68 | 7.17 | 61.23 | 70.01 | 13.36 | |
| 13 | 43.44 | 7.17 | 61.23 | 70.01 | 13.36 | |

Showing 1 to 14 of 176 entries

# Subsetting data

- In this module, we will explore only a subset of this dataset, which includes the following variables:
  - yield
  - 3 material variables
  - 3 process variables

CMP ×

Filter

| | Yield | BiologicalMaterial01 | BiologicalMaterial02 | BiologicalMaterial03 |
|---|---|---|---|---|
| 1 | 38.00 | 6.25 | 49.58 | 56.97 |
| 2 | 42.44 | 8.01 | 60.97 | 67.48 |
| 3 | 42.03 | 8.01 | 60.97 | 67.48 |
| 4 | 41.42 | 8.01 | 60.97 | 67.48 |
| 5 | 42.49 | 7.47 | 63.33 | 72.25 |
| 6 | 43.57 | 6.12 | 58.36 | 65.31 |
| 7 | 43.12 | 7.48 | 64.47 | 72.41 |
| 8 | 43.06 | 6.94 | 63.60 | 72.06 |
| 9 | 41.49 | 6.94 | 63.60 | 72.06 |

Showing 1 to 10 of 176 entries

...

CMP ×

Filter

| ManufacturingProcess01 | ManufacturingProcess02 | ManufacturingProcess03 |
|---|---|---|
| NA | NA | NA |
| 0.0 | 0.0 | NA |
| 0.0 | 0.0 | NA |
| 0.0 | 0.0 | NA |
| 10.7 | 0.0 | NA |
| 12.0 | 0.0 | NA |
| 11.5 | 0.0 | 1.56 |
| 12.0 | 0.0 | 1.55 |
| 12.0 | 0.0 | 1.56 |

Showing 1 to 10 of 176 entries

# Subsetting data (cont'd)

```
# Let's make a vector of column indices we would like to save.
column_ids = c(1:4,  #<- concatenate a range of ids
               14:16)#<- with another a range of ids
column_ids
```

```
[1]  1  2  3  4 14 15 16
```

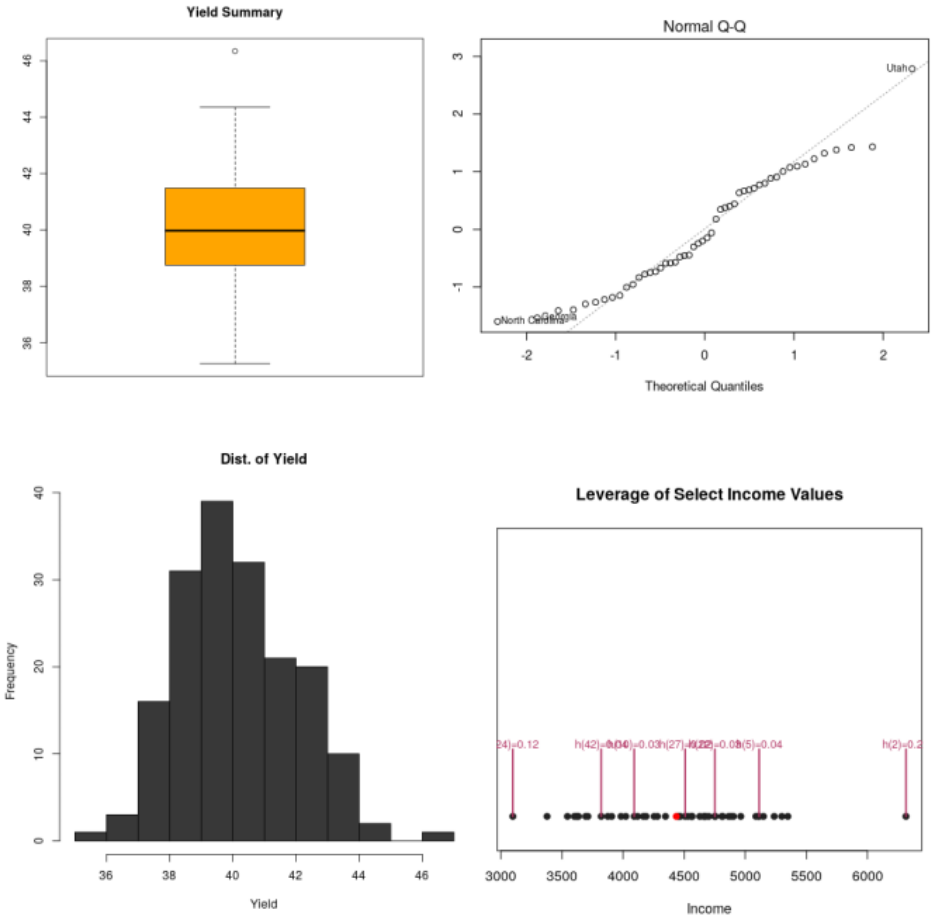
```
# Let's save the subset into a new variable.
CMP_subset = CMP[ , column_ids]
str(CMP_subset)
```

```
'data.frame':   176 obs. of  7 variables:
 $ Yield                 : num  38 42.4 42 41.4 42.5 ...
 $ BiologicalMaterial01  : num  6.25 8.01 8.01 8.01 7.47 6.12 7.48 6.94 6.94 6.94 ...
 $ BiologicalMaterial02  : num  49.6 61 61 61 63.3 ...
 $ BiologicalMaterial03  : num  57 67.5 67.5 67.5 72.2 ...
 $ ManufacturingProcess01: num  NA 0 0 0 10.7 12 11.5 12 12 12 ...
 $ ManufacturingProcess02: num  NA 0 0 0 0 0 0 0 0 0 ...
 $ ManufacturingProcess03: num  NA NA NA NA NA NA 1.56 1.55 1.56 1.55 ...
```

# Univariate plots

- Univariate plots are used to visualize distribution of a **single variable**
- They are used primarily in the initial stages of EDA when we would like to learn more about individual variables in our data
- They are also used in combination with other univariate plots to compare data distributions of different variables
- Univariate plots include the following popular graphs: `boxplot`, `histogram`, `density curve`, `dot plot`, `QQ plot`, and `bar plot`

## Different univariate plots



# Univariate plots: boxplots

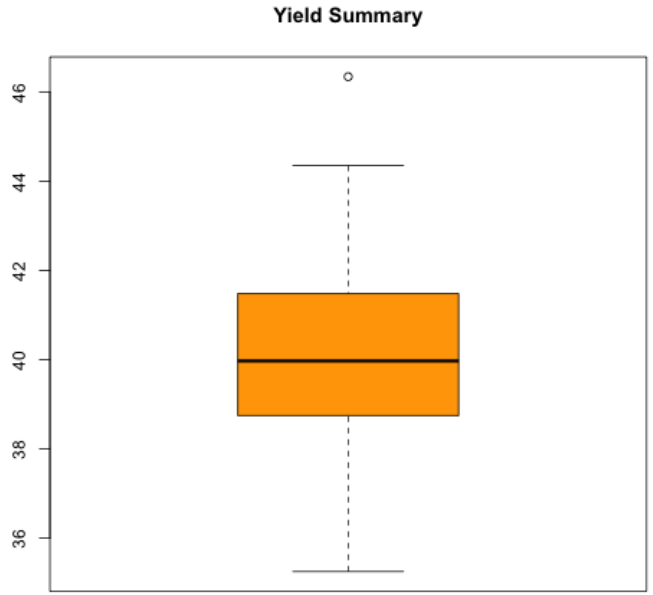
- A box-and-whisker plot (a.k.a. boxplot) is a graph that depicts the distribution of a continuous variable through its five-point summary:

```
summary(CMP_subset$Yield)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  35.25   38.75   39.97   40.18   41.48   46.34 
```

- **Min value** (`35.25`): bottom `whisker` of the boxplot
- **Max value** (`46.34`): circle (it's an outlier!)
- **1st quartile value** (`38.75`): bottom of the box
- **Median (2nd quartile) value** (`39.97`): line in the box
- **3rd quartile value** (`41.48`): top of the box

```
boxplot(CMP_subset$Yield,
        col = "orange",
        main = "Yield Summary")
#<- add title
```

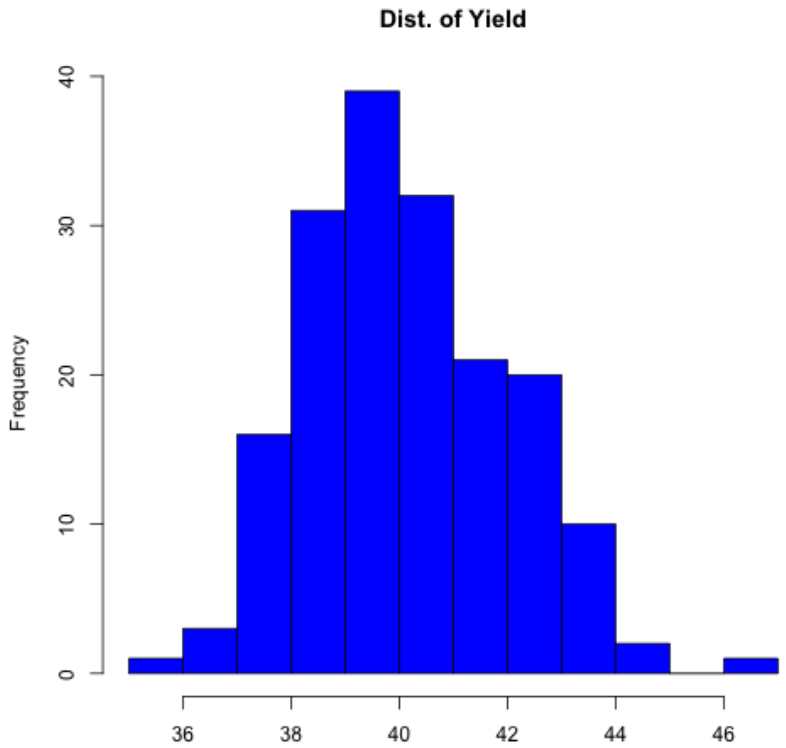


# Univariate plots: histogram

- A histogram represents the **distribution of numerical data**
  - The height of each bar has been calculated as the number of observations in that range
  - `hist()` produces a basic histogram of any *numeric* variable
- Let's plot a histogram of `Yield`

## Graphical summary

```
# Univariate plot: histogram.
hist(CMP_subset$Yield,
     col = "blue",
     xlab = "Yield",   #<- set x-axis label
     main = "Dist. of Yield") #<- set title
```



# Univariate plots: histogram (cont'd)

We can also look at just the numerical summary of a histogram by setting `plot = FALSE`

```
# Histogram data without plot.
hist(CMP_subset$Yield, plot = FALSE)
```

## Numerical summary

```
$breaks
 [1] 35 36 37 38 39 40 41 42 43 44 45 46 47

$counts
 [1]  1  3 16 31 39 32 21 20 10  2  0  1

$density
 [1] 0.005681818 0.017045455 0.090909091
0.176136364 0.221590909 0.181818182
 [7] 0.119318182 0.113636364 0.056818182
0.011363636 0.000000000 0.005681818

$mids
 [1] 35.5 36.5 37.5 38.5 39.5 40.5 41.5 42.5
43.5 44.5 45.5 46.5

$xname
[1] "CMP_subset$Yield"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
```

# Knowledge Check 3

# Exercise 3

# Visualizations


We have just begun to touch on visualizations in R, but there are many more **univariate, bivariate and multivariate** plots to explore

Take a look at all that's possible as you build on the basics of ggplot 2 here:

***https://www.r-graph-gallery.com/ggplot2-package.html***

# Module completion checklist

| Topic | Complete |
|---|---|
| Select specific variables, sometimes using specific rules, using the select command | ✔ |
| Derive new variables from the existing variables using the mutate and transmute commands | ✔ |
| Summarize columns using the summarise and group by functions | ✔ |
| Understand tidy data and its advantages | ✔ |
| Manipulate columns by using the separate and unite functions | ✔ |
| Define the exploratory data analysis (EDA) cycle and the differences between static and interactive visualizations | ✔ |
| Describe and build univariate plots to illustrate patterns in data | ✔ |

# Summary

- In today's module, we learned about the **`tidyr`** package in `tidyverse`
- With this package, we were able to tidy up datasets in a more elegant way
- After class, you can perform your own transformations with other built-in R datasets

# Next steps

- Check out our Text mining course in the last week of August. It will cover:
  - Dealing with text data
  - Concepts of Natural Language processing and text mining
  - Cleaning and visualizing text data
  - Perform topic modelling using LDA

# This completes our module

**Congratulations!**