

CS&SS 321 - Data Science and Statistics for Social Sciences

**Module II - Data management and exploratory visual
analysis**

Ramses Llobet

Module II

- ▶ This model will equip students with essential data science skills in R.
- ▶ In the next quiz sections, we will cover the following topics:
 - ▶ **Data frames, logical** relations, and **subsetting**.
 - ▶ **Quantile** and NA data.
 - ▶ Base R graphics.
 - ▶ **Pivoting** and **merging** data.
 - ▶ Introduction to ggplot2.

Creating and manipulating data frames.

- Think about *data* in terms of **data frame**.

“**TIDY DATA** is a standard way of mapping the meaning of a dataset to its structure.”

—HADLEY WICKHAM

In tidy data:

- each **variable** forms a **column**
- each **observation** forms a **row**
- each **cell** is a **single measurement**

each column a variable

id	name	color
1	floof	gray
2	max	black
3	cat	orange
4	donut	gray
5	merlin	black
6	panda	calico

each row an observation

Wickham, H. (2014). Tidy Data. Journal of Statistical Software 59 (10). DOI: 10.18637/jss.v059.i10

Creating and manipulating data frames.

- ▶ A data frame is a special type of object in R that can store **multiple vectors** of data.
- ▶ We can create data frames using the function `data.frame()`.

```
# vectors with student's names and grades
student <- c("Alice", "Bob", "Charlie","Sean","Brandy")
grades_M <- c(76, 82, 94, 45, 75)
grades_F <- c(82, 90, 89, NA, 64)

# create a df with grades
(df_new <- data.frame(student,grades_M,grades_F))
```

```
##   student grades_M grades_F
## 1   Alice      76      82
## 2    Bob      82      90
## 3 Charlie      94      89
## 4    Sean      45      NA
## 5 Brandy      75      64
```

Creating and manipulating data frames.

- We can create data frames by directly writing the vectors/columns as separate elements within the `data.frame()` function:

```
df_new <- data.frame(student=c("Alice", "Bob", "Charlie",  
                              "Sean", "Brandy"),  
                     grades_M=c(76, 82, 94, 45, 75),  
                     grades_F=c(82, 90, 89, NA, 64))
```

```
df_new
```

```
##   student grades_M grades_F  
## 1   Alice      76      82  
## 2    Bob      82      90  
## 3 Charlie      94      89  
## 4   Sean      45     NA  
## 5 Brandy      75      64
```

Creating and manipulating data frames.

- ▶ To select a specific column from a data frame, use the \$ operator followed by the *name* of the column.

```
df_new$grades_M
```

```
## [1] 76 82 94 45 75
```

- ▶ To select specific rows and/or columns from a data frame, we use brackets [].
- ▶ If the object is a single vector, we use a single numeric value in the brackets to select an element within the vector.

```
# select element 2 from vector grade_M:  
grades_M[2]
```

```
## [1] 82
```

Creating and manipulating data frames.

- ▶ If the object is a matrix or data frame, we can select elements by their row and column positions.
 - ▶ **Note:** we input two different values separated by a **comma** to select the row and column

```
# select row 2 from object df:  
df_new[2,]
```

```
##      student grades_M grades_F  
## 2      Bob      82      90
```

```
# select column 2 from object df:  
df_new[,2]
```

```
## [1] 76 82 94 45 75
```

```
# select element in row 2 and column 2:  
df_new[2,2]
```

Creating and manipulating data frames.

- We can also use **characters** to select columns by their names, for example:

```
# select column name "grade_M" from object df:
```

```
df_new[,c("grades_M")]
```

```
## [1] 76 82 94 45 75
```

```
# select columns name "student" and "grade_M":
```

```
df_new[,c("student","grades_M")]
```

```
##   student grades_M
## 1   Alice       76
## 2    Bob       82
## 3 Charlie       94
## 4   Sean       45
## 5 Brandy       75
```


tibbles are data frames too!

- ▶ Another type of data frame are **tibbles**.
 - ▶ `tibble()` is a fancy version of `data.frame()`.
 - ▶ All dplyr functions provide outputs as `as_tibbles`.

```
as_tibble(df_new) ; class(as_tibble(df_new))
```

```
## # A tibble: 5 x 3
##   student grades_M grades_F
##   <chr>      <dbl>    <dbl>
## 1 Alice      76      82
## 2 Bob        82      90
## 3 Charlie    94      89
## 4 Sean       45      NA
## 5 Brandy     75      64
```

```
## [1] "tbl_df"      "tbl"         "data.frame"
```

Logical relations

► Logical Data Class:

- Represents binary values: TRUE or FALSE.
- Can be transformed into numeric form: TRUE becomes 1, and FALSE becomes 0.
- Useful for relational analyses and evaluating proportions of TRUE within a vector using the `mean()` function.
- Used to set **conditional tests**; useful for **subsetting** or create new variables.

```
3 + 5 < 10 # is 3 + 5 less than 10?
```

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

Logical relations

```
# select column name "grade_M" from object df:  
df_new$grades_M
```

```
## [1] 76 82 94 45 75
```

```
# Is each value greater or equal to 80?  
df_new$grades_M >= 80 # the condition ">= 80" sets a logical test
```

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

```
# What proportion of TRUEs are in this vector?  
mean(df_new$grades_M >= 80) # `TRUE` == 1, and `FALSE` == 0
```

```
## [1] 0.4
```

Subsetting: `ifelse()`.

- ▶ We can use the `ifelse()` function to create new variables based on *conditions* from other variables.
 - 1 - We set a *logical test* that evaluates to TRUE or FALSE.
 - 2 - We specify what value to assign if the test is TRUE, and a different value if the test is FALSE.

```
# if test is TRUE, then "pass", otherwise, then "fail"
df_new$midterm <- ifelse(df_new$grades_M > 60, "pass", "fail")
```

```
df_new
```

##	student	grades_M	grades_F	midterm
## 1	Alice	76	82	pass
## 2	Bob	82	90	pass
## 3	Charlie	94	89	pass
## 4	Sean	45	NA	fail
## 5	Brandy	75	64	pass

Subsetting: Base R.

- We can use *logical tests* in **vectors** within the **row element** of an object `x[test ,]` to subset those cases that are TRUE.

```
# In the vector midterm, what values are "pass"?  
df_new$midterm=="pass"
```

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

```
# subset those rows where this test is TRUE  
df_new[ df_new$midterm=="pass" , ]
```

```
## student grades_M grades_F midterm  
## 1 Alice 76 82 pass  
## 2 Bob 82 90 pass  
## 3 Charlie 94 89 pass  
## 5 Brandy 75 64 pass
```

Subsetting: `subset()`/`filter()`.

- ▶ To subset data, we can use the functions `subset()` or `filter()`.
 - ▶ The `subset()` function is part of base R, while `filter()` is a function from the `dplyr` package.
 - ▶ If you plan to use `filter()`, you need to load the `tidyverse` or `dplyr` package first.

subset the df into a new one with final exam grades of above 85

```
subset(df_new, grades_F > 85)
```

```
##   student grades_M grades_F midterm
## 2      Bob      82      90    pass
## 3 Charlie      94      89    pass
```

filter(df_new, grades_F > 85) # from dplyr package

```
##   student grades_M grades_F midterm
## 1      Bob      82      90    pass
## 2 Charlie      94      89    pass
```

Processing NA data.

- An initial step in data science project analysis is to examine the NA values.

```
dat
```

##		name	age	gender	score
## 1		Alice	20	F	85
## 2		Bob	30	M	62
## 3		Charlie	NA	M	75
## 4		Dave	28	M	80
## 5		Eve	22	F	95
## 6		Marta	21	F	NA

Processing NA data.

- ▶ The function `is.na()` will return a vector of logical values

```
is.na(dat)
```

```
##      name  age gender score
## [1,] FALSE FALSE FALSE FALSE
## [2,] FALSE FALSE FALSE FALSE
## [3,] FALSE  TRUE FALSE FALSE
## [4,] FALSE FALSE FALSE FALSE
## [5,] FALSE FALSE FALSE FALSE
## [6,] FALSE FALSE FALSE  TRUE
```

```
mean(is.na(dat))
```

```
## [1] 0.08333333
```


Processing NA data.

- ▶ Several packages have functions to assist the analysis of NA values.
 - ▶ function `freq.na()` from package `questionr` is an example:

```
library(questionr)
freq.na(dat)
```

```
##           missing %
## age             1 17
## score           1 17
## name            0  0
## gender          0  0
```

Processing NA data.

- We already know that some functions have the argument `na.rm`, but this is not the norm.

```
dat$score
```

```
## [1] 85 62 75 80 95 NA
```

```
mean(dat$score)
```

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

```
mean(dat$score, na.rm = TRUE)
```

```
## [1] 79.4
```

Processing NA data.

- The `na.omit()` function in base R removes all rows with any NA value.

```
dat
```

```
##      name age gender score
## 1  Alice  20      F    85
## 2   Bob   30      M    62
## 3 Charlie NA      M    75
## 4   Dave  28      M    80
## 5    Eve  22      F    95
## 6  Marta  21      F    NA
```

```
na.omit(dat)
```

```
##      name age gender score
## 1  Alice  20      F    85
## 2   Bob   30      M    62
```

Processing NA data.

- The `drop_na()` function from `dplyr` removes all rows with any NA value of a specific column.

```
drop_na(dat, score)
```

##		name	age	gender	score
## 1		Alice	20	F	85
## 2		Bob	30	M	62
## 3		Charlie	NA	M	75
## 4		Dave	28	M	80
## 5		Eve	22	F	95

```
drop_na(dat, age)
```

##		name	age	gender	score
## 1		Alice	20	F	85
## 2		Bob	30	M	62
## 3		Dave	28	M	80
## 4		Eve	22	F	95
## 5		Marta	21	F	NA

Processing NA data.

- We can use `ifelse()` function to substitute NA values.

```
dat$score <- ifelse( is.na( dat$score ),  
                    0, # if TRUE  
                    dat$score) # if FALSE
```

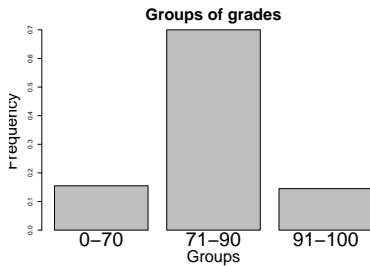
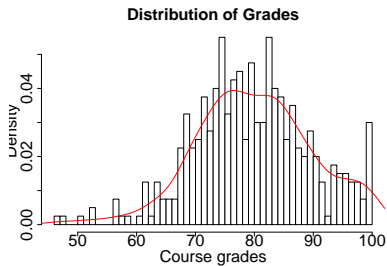
dat

##	name	age	gender	score
## 1	Alice	20	F	85
## 2	Bob	30	M	62
## 3	Charlie	NA	M	75
## 4	Dave	28	M	80
## 5	Eve	22	F	95
## 6	Marta	21	F	0

Distributions

- ▶ A **distribution** describes how variable values are **spread across** possible outcomes.
 - ▶ A **probability** distribution represents the **likelihood** of specific outcomes.
 - ▶ A **frequency** distribution summarizes counts of **distinct** values or ranges in dataset.
- ▶ **Continuous vs. Discrete Distributions:**
 - ▶ **Continuous** distributions involve numerical variables that can take any value within a range (e.g., height, weight), while
 - ▶ **Discrete** distributions involve variables that take distinct, separate values (e.g., number of cars, number of people).

Continuous vs. Discrete Distributions



Data Generating Process

- ▶ A **Data Generating Process** (DGP) refers to the hypothetical or real mechanism that generates a dataset.
 - ▶ It is a conceptual model that describes **how** the observed data is generated or produced.
- ▶ **Distributions** represent **systematic behavior** (aka, DGP).
- ▶ When looking at a distributions:
 - ▶ think in terms of a **DGP**, and
 - ▶ **how** the data was generated.

Data Generating Process

- ▶ Two very useful pieces of information from a DGP are its **mean** and **standard deviation**.

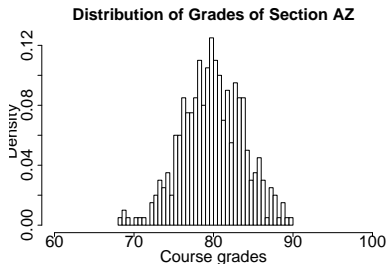
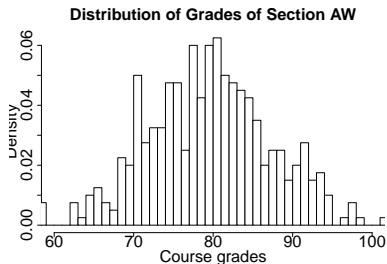
$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i \quad ; \quad S = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2}$$

where

- ▶ \bar{X} represents the **sample mean**.
- ▶ n is the number of **observations** in the sample.
- ▶ X_i represents **values** from a variable in the sample.
- ▶ S represents the **sample standard deviation**.

Data Generating Process: standard deviation

- The **standard deviation** gives us information about how spread is the data around the mean.



Reporting distributions

- ▶ When analyzing data, always report **descriptive statistics**.
 - ▶ *Mean.*
 - ▶ *Median.*
 - ▶ *Standard deviation.*
 - ▶ *Minimum.*
 - ▶ *Maximum.*
 - ▶ *Quartiles.*
- ▶ Note:
 - ▶ When comparing distributions of the same quantities, use the **median** instead of the **mean** as the reference point. **Why?**

quantile and data distribution.

- The quantile function in R can be used to calculate the values that separate a distribution into different quantiles.

```
quantile(df$grades)
```

```
##      0%      25%      50%      75%     100%  
## 46.00  74.00  80.00  86.25 100.00
```

```
quantile(df$grades, probs = c(0.25, 0.5, 0.75))
```

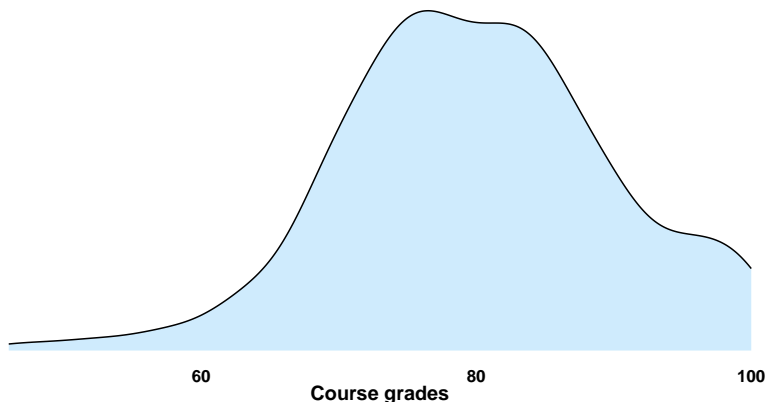
```
##      25%      50%      75%  
## 74.00 80.00 86.25
```

```
summary(df$grades)
```

```
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.  
## 46.00   74.00   80.00   79.98  86.25  100.00
```

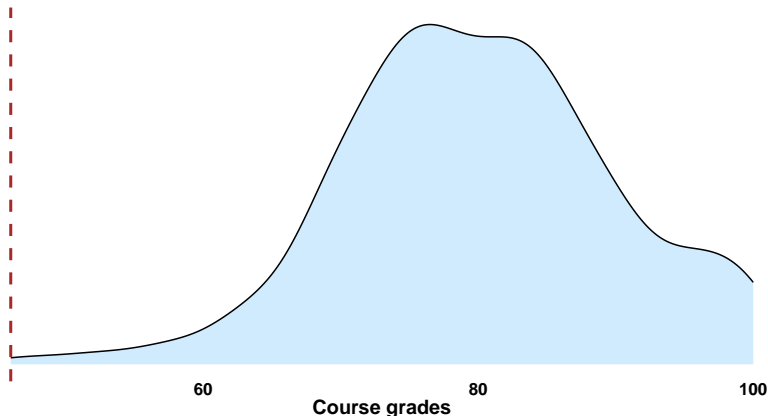
quantile and data distribution.

- ▶ Visualizing quantiles.
- ▶ Use the argument `probs` to specify segments of the data.



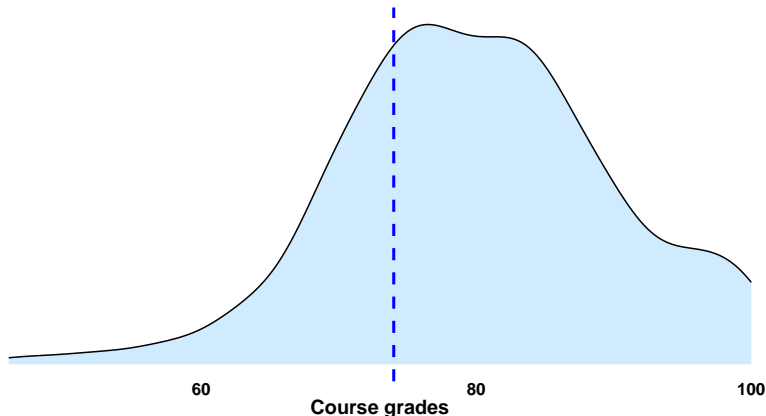
quantile and data distribution.

- ▶ Visualizing quantiles: **minimum**.
- ▶ `quantile(df$x, probs = 0)`



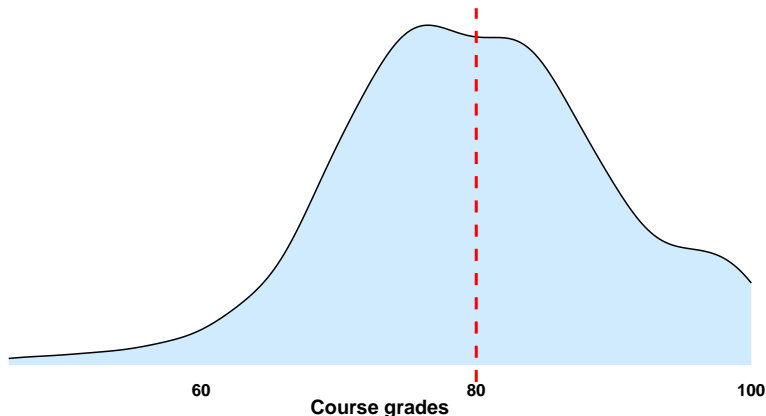
quantile and data distribution.

- ▶ Visualizing quantiles: **1st Quartile (Q1)** or **25th Percentile**.
- ▶ `quantile(df$x, probs = 0.25)`



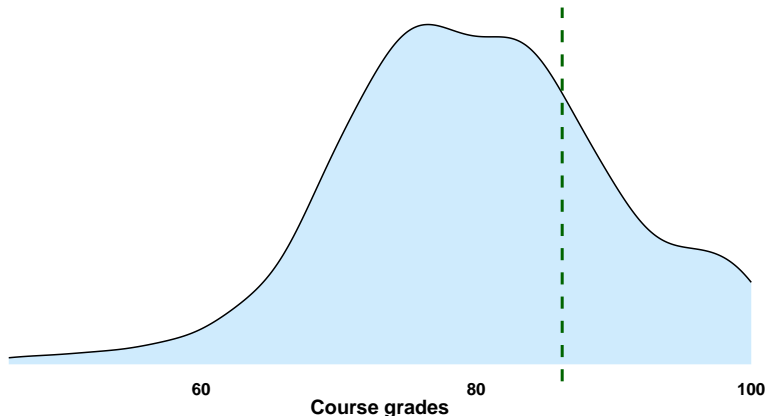
quantile and data distribution.

- ▶ Visualizing quantiles: **2st Quartile (Q2)** or **50th Percentile** or **median** or **5th Decile**.
- ▶ `quantile(df$x, probs = 0.5)`



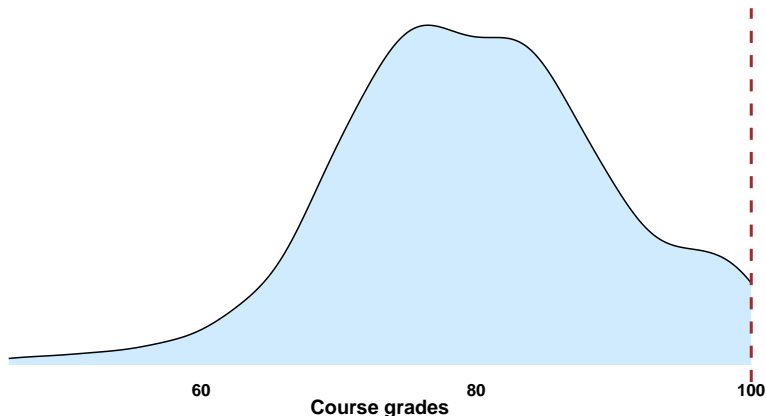
quantile and data distribution.

- ▶ Visualizing quantiles: **3st Quartile (Q3)** or **75th Percentile**.
- ▶ `quantile(df$x, probs = 0.75)`



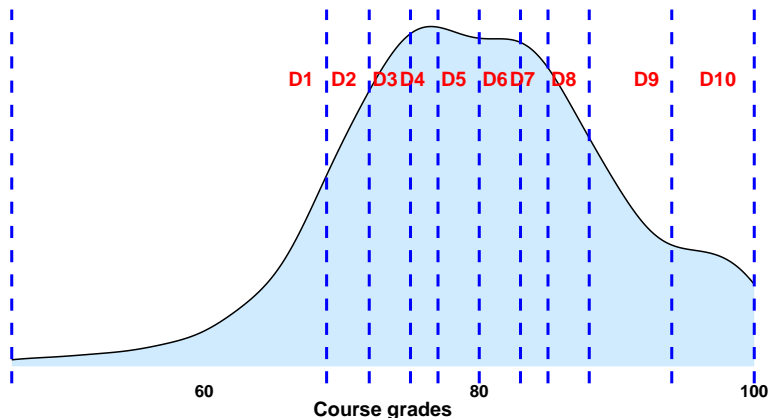
quantile and data distribution.

- ▶ Visualizaing quantiles: **maximum** or **100th percentile** or **10th decile**.
- ▶ `quantile(df$x, probs = 1)`



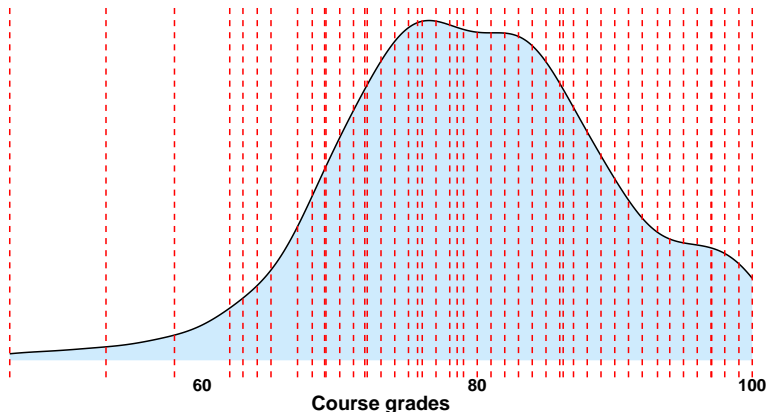
quantile and data distribution.

- ▶ Visualizing quantiles: **deciles** (1-10).
- ▶ `quantile(df$x, probs = seq(from=0,to=1,by=0.1))`



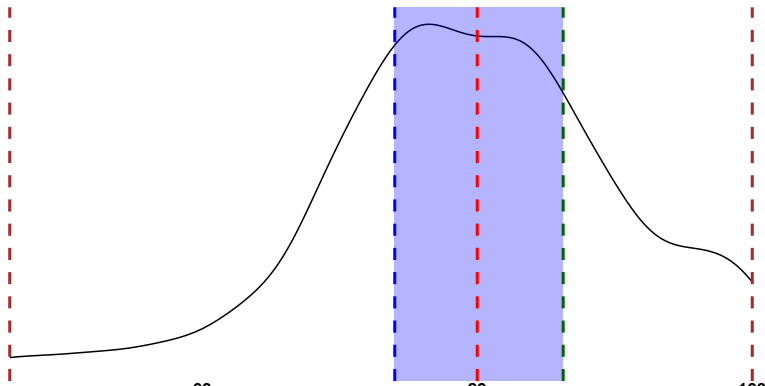
quantile and data distribution.

- ▶ Visualizing quantiles: **percentiles** (1-100).
- ▶ `quantile(df$x, probs = seq(from=0,to=1,by=0.01))`



quantile and data distribution.

- ▶ The **interquartile range** (IQR) is a measure of variability that represents the difference between the **first** and the **third** quartiles.
- ▶ It provides information about the spread of the middle 50% of the data.



More functions: nested ifelse.

- To create a new discrete variable, *letter*, with three levels (C, B, and A) based on exam scores, consider using *ifelse*.

```
dat$letter <- ifelse(dat$score < 70,  
                     "C", # if TRUE  
                     "Otherwise") # if FALSE  
dat
```

##	name	age	gender	score	letter
## 1	Alice	20	F	85	Otherwise
## 2	Bob	30	M	62	C
## 3	Charlie	NA	M	75	Otherwise
## 4	Dave	28	M	80	Otherwise
## 5	Eve	22	F	95	Otherwise
## 6	Marta	21	F	0	C

More functions: nested ifelse.

- However, note that `ifelse` yields binary results determined by the conditional test's TRUE or FALSE.

```
dat$letter <- ifelse(dat$score >= 70 & dat$score < 85,  
                    "B", # if TRUE  
                    "Otherwise") # if FALSE  
dat
```

##	name	age	gender	score	letter
## 1	Alice	20	F	85	Otherwise
## 2	Bob	30	M	62	Otherwise
## 3	Charlie	NA	M	75	B
## 4	Dave	28	M	80	B
## 5	Eve	22	F	95	Otherwise
## 6	Marta	21	F	0	Otherwise

More functions: nested ifelse.

- Can we do better and use ifelse to map several characters into a vector using conditional tests?

```
dat$letter <- ifelse(dat$score >= 85,  
                     "A", # if TRUE  
                     "Otherwise") # if FALSE  
dat
```

##	name	age	gender	score	letter
## 1	Alice	20	F	85	A
## 2	Bob	30	M	62	Otherwise
## 3	Charlie	NA	M	75	Otherwise
## 4	Dave	28	M	80	Otherwise
## 5	Eve	22	F	95	A
## 6	Marta	21	F	0	Otherwise

More functions: nested ifelse.

- Yes! ifelse function can be nested on itself for multiple tests.

```
dat$letter <- ifelse(dat$score < 70,  
                    "C", # if TRUE  
                    ifelse(dat$score >= 70 & dat$score < 85, # if FALSE  
                            "B", # if TRUE  
                            ifelse(dat$score >= 85, # if FALSE  
                                    "A", # if TRUE  
                                    NA))) # if FALSE  
dat
```

##		name	age	gender	score	letter
## 1		Alice	20	F	85	A
## 2		Bob	30	M	62	C
## 3		Charlie	NA	M	75	B
## 4		Dave	28	M	80	B
## 5		Eve	22	F	95	A
## 6		Marta	21	F	0	C

More functions: nested case_when.

- ▶ You can use the `case_when` function from the `dplyr` package to produce the same output.

```
dat$letter <- case_when(dat$score < 70 ~ "C",  
                        dat$score >= 70 & dat$score < 85 ~ "B",  
                        dat$score >= 85 ~ "A")
```

dat

##	name	age	gender	score	letter
## 1	Alice	20	F	85	A
## 2	Bob	30	M	62	C
## 3	Charlie	NA	M	75	B
## 4	Dave	28	M	80	B
## 5	Eve	22	F	95	A
## 6	Marta	21	F	0	C

Data class: factors

- ▶ **Categorical variables** can take on a limited, and usually fixed, number of different values or levels.
 - ▶ Voted:
 - ▶ Yes/No
 - ▶ Political parties:
 - ▶ Social democrat, Liberals, Conservatives, Green party, etc
 - ▶ *Likert scales* in survey opinions:
 - ▶ Strongly Agree, Agree, Disagree, Strongly Disagree
- ▶ However, `character` data type in R is used to store sequences of characters (text).

Factors

- ▶ A factor is a data structure used to represent categorical variables.

```
gender <- c("Male", "Female", "Male", "Female")  
  
class(gender)
```

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

```
gender_factor <- as.factor(gender)  
  
class(gender_factor)
```

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

Factors: levels

- ▶ Factors have levels, which are the distinct values that the categorical variable can take.
- ▶ The levels are determined by the unique values in the original vector.

```
# Checking levels of a factor  
levels(gender_factor)
```

```
## [1] "Female" "Male"
```

Factors: Ordering Levels

By default, levels are ordered alphabetically. You can customize the order using the `levels` argument.

```
# Example: Specifying custom order  
(ordered_gender <- factor(gender,  
                           levels = c("Male", "Female")))
```

```
## [1] Male   Female Male   Female  
## Levels: Male Female
```

```
# Checking the levels of a facto variable  
levels(ordered_gender)
```

```
## [1] "Male"    "Female"
```

Good practice: create factor variables

- ▶ Some functions, especially in ggplot2 for visualization, require factors to function properly.
- ▶ It is a good practice to create new variables as **factors** when initiating an analysis.

```
(female <- c(0,1,0,1,0,0))
```

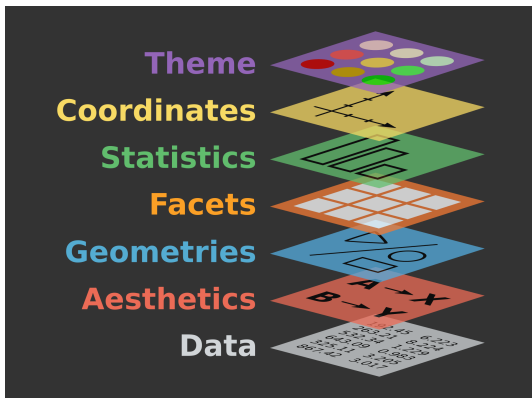
```
## [1] 0 1 0 1 0 0
```

```
(female_f <- factor(female,  
                    levels = c(0,1), # map the levels order  
                    labels = c("male","female")))
```

```
## [1] male   female male   female male   male  
## Levels: male female
```

Grammar of graphics

- A statistical graphic is a mapping of data variables to aesthetic attributes of geometric objects. (Wilkinson 2005)



Grammar of graphics in ggplot2

- ▶ ggplot2: A *layered* grammar of graphics ([Wickham 2009](#)).
 - ▶ Build a graphic from multiple layers; each consists of some geometric objects or transformation
 - ▶ Use + to stack up layers
- ▶ *What* data do you want to visualize?
 - ▶ `ggplot(data = ...)`
- ▶ *How* are variables mapped to specific aesthetic attributes?
 - ▶ `aes(... = ...)`
 - ▶ positions (x, y), shape, colour, size, fill, alpha, linetype, label...
 - ▶ If the value of an attribute do not vary w.r.t. some variable, don't wrap it within `aes(...)`
- ▶ *Which* geometric shapes do you use to represent the data?
 - ▶ `geom_{}`:
 - ▶ `geom_point`, `geom_line`, `geom_ribbon`, `geom_polygon`, `geom_label`...

Tidy data

- ▶ `ggplot2` works well only with tidy data
 - ▶ *Tidy data*:
 - ▶ Each **variable** must have its own **column**
 - ▶ Each **observation** must have its own **row**
 - ▶ Each value must have its own cell

Intro to ggplot

- ▶ How to create a **scatter plot**: *continuous* vs. *continuous* variables
- ▶ How to create a **boxplot**: *continuous* vs **categorical** variables

```
summary(mtcars[,c("mpg", "wt", "cyl")])
```

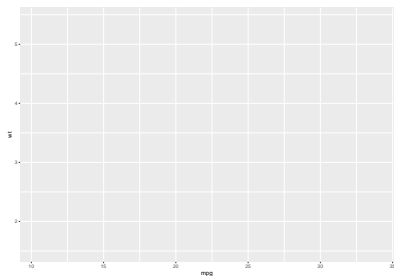
##	mpg	wt	cyl
##	Min. :10.40	Min. :1.513	Min. :4.000
##	1st Qu.:15.43	1st Qu.:2.581	1st Qu.:4.000
##	Median :19.20	Median :3.325	Median :6.000
##	Mean :20.09	Mean :3.217	Mean :6.188
##	3rd Qu.:22.80	3rd Qu.:3.610	3rd Qu.:8.000
##	Max. :33.90	Max. :5.424	Max. :8.000

Building a plot from scratch

Step 1: Define a basic ggplot object with x and y aesthetics

```
library(ggplot2)

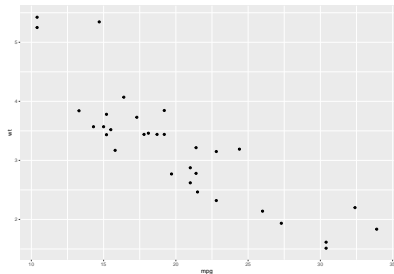
ggplot(data=mtcars,
       aes(x=mpg,
           y=wt))
```



Building a plot from scratch: scatter plot

Step 2: Define a geometric shape

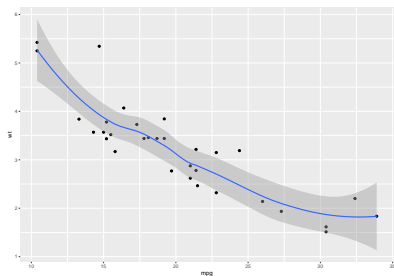
```
ggplot(data=mtcars,  
       aes(x=mpg,  
           y=wt)) +  
  geom_point()
```



Building a plot from scratch: scatter plot

Note: we are not limited to have a single geometric form,

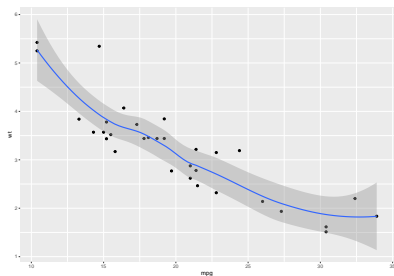
```
ggplot(data=mtcars,  
       aes(x=mpg,  
           y=wt)) +  
  geom_point() +  
  geom_smooth()
```



Building a plot from scratch: scatter plot

Note: we are not limited to have a single geometric form,

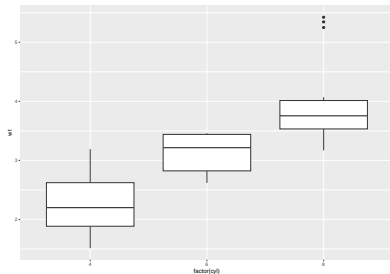
```
ggplot(data=mtcars,  
       aes(x=mpg,  
           y=wt)) +  
  geom_point() +  
  geom_smooth()
```



Building a plot from scratch: boxplot

Note: we are not limited to have a single geometric form,

```
ggplot(data=mtcars,  
       aes(x=factor(cyl),  
           y=wt)) +  
  geom_boxplot()
```



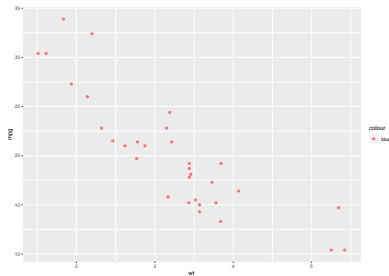
More on ggplot: colors

- ▶ In ggplot, the **color** aesthetic is used to group data by a categorical or numerical variable, with each group automatically assigned a unique color.
- ▶ Customize the color aesthetic using the argument in a geom shape, you cannot manually fit **colors** in the `aes()` function because ggplot will assume that you are fitting a **factor variable**.
- ▶ Use the `scale_color_manual()` function to set specific colors for values in a plot by passing a named vector of colors to the `values` argument.
- ▶ You can create your own custom color palette with the `scale_color_manual()` function by specifying a named vector of colors.

More on ggplot: colors

- Why the following plot is not blue?

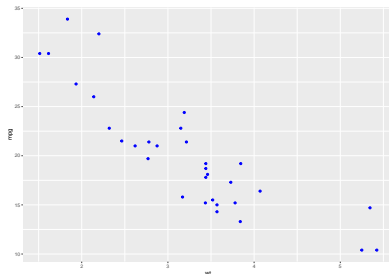
```
ggplot(mtcars, aes(x = wt,  
                  y = mpg,  
                  color="blue")) +  
  geom_point()
```



More on ggplot: colors

- Customize the color aesthetic using the argument in a geom shape, you cannot manually fit **colors** in the aes() function because ggplot will assume that you are fitting a **factor variable**.

```
ggplot(mtcars, aes(x = wt,  
                   y = mpg)) +  
  geom_point(color = "blue")
```



More on ggplot: colors

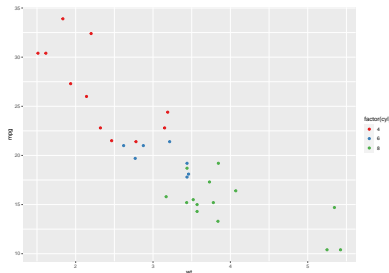
```
ggplot(mtcars, aes(x = wt,
                   y = mpg,
                   color = factor(cyl)))
  geom_point()
```

More on ggplot: colors

- Built-in ggplot color **palettes** can be used by passing the name of the palette to the palette argument of the `scale_color_()` function. For instance, `scale_color_brewer()` can be used to apply a palette from the **ColorBrewer** library.

```
library(RColorBrewer)

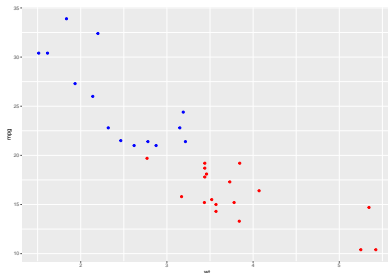
ggplot(mtcars, aes(x = wt,
                  y = mpg,
                  color = factor(cyl)))
  geom_point() +
  scale_color_brewer(palette = "Set1")
```



More on ggplot: colors

- You can customize which elements shall be colored using `filter`

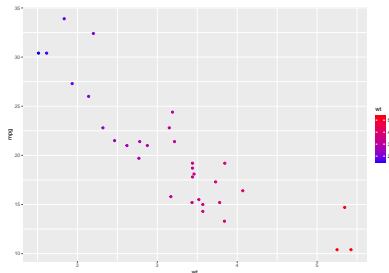
```
ggplot(mtcars, aes(x = wt,  
                   y = mpg)) +  
  geom_point(color =  
             ifelse(mtcars$mpg > 20,  
                    "blue", "red"))
```



More on ggplot: colors

- You can also fit numerical variables or manually scale the gradient between several colors.

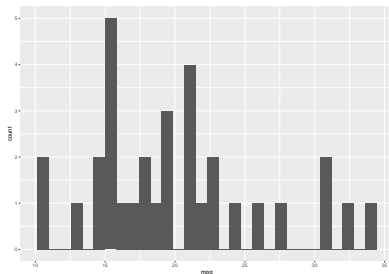
```
ggplot(mtcars, aes(x = wt,  
                   y = mpg,  
                   color = wt)) +  
  geom_point() +  
  scale_color_gradient(low = "blue",  
                      high = "red")
```



More on ggplot: histograms

- When defining histograms or barplots, you only need to define the x aesthetics as y becomes the relative or absolute frequencies

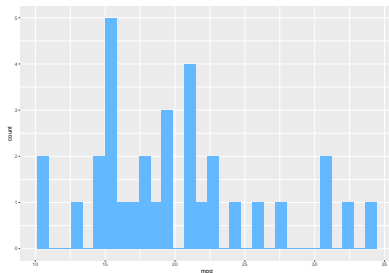
```
ggplot(mtcars,  
       aes(x = mpg)) +  
  geom_histogram()
```



More on ggplot: histograms

- To color geometries with large areas, you will need to use the `fill=` argument.

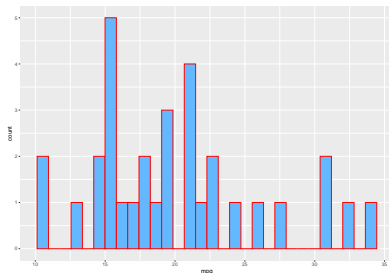
```
ggplot(mtcars,  
  aes(x = mpg)) +  
  geom_histogram(fill="steelblue1")
```



More on ggplot: histograms

- The argument `color` will provide color in the surface of the geometry.

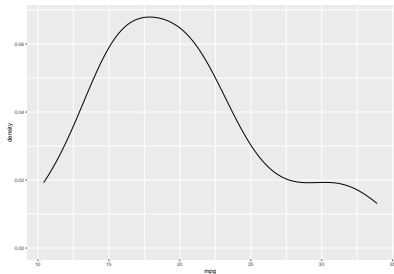
```
ggplot(mtcars,
       aes(x = mpg)) +
  geom_histogram(fill="steelblue1",
                color="red")
```



More on ggplot: density plots

- For a density plots, use `geom_density()`.

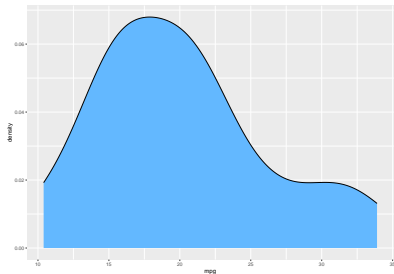
```
ggplot(mtcars,  
       aes(x = mpg)) +  
  geom_density()
```



More on ggplot: density plots

- For a density plots, use `geom_density`.

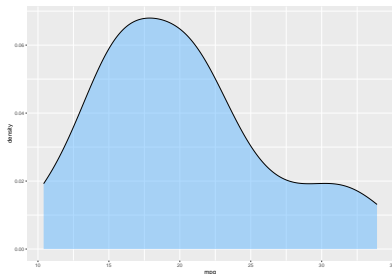
```
ggplot(mtcars,  
       aes(x = mpg)) +  
  geom_density(fill="steelblue1")
```



More on ggplot: density plots

- ▶ Geometric shapes have `alpha=` which controls the transparency of the geometry.
 - ▶ `alpha` values range from 0 (transparent) to 1 (opaque).

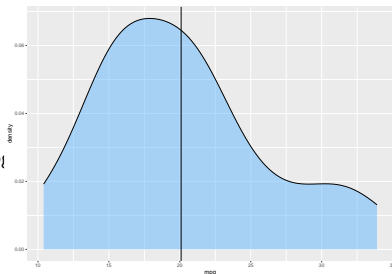
```
ggplot(mtcars,  
  aes(x = mpg)) +  
  geom_density(fill="steelblue1",  
    alpha=0.5)
```



More on ggplot: density plots

- You can use the geometry `geom_vline` to include vertical lines and highlight points or thresholds of interest, like the mean

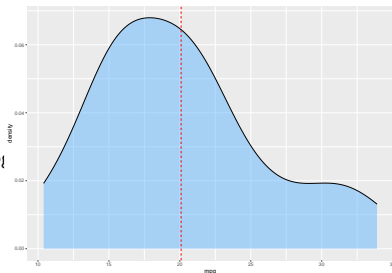
```
ggplot(mtcars,
       aes(x = mpg)) +
  geom_density(fill="steelblue1",
              alpha=0.5) +
  geom_vline(xintercept = mean(mtcars$mpg))
```



More on ggplot: density plots

- The geometry `geom_vline` can be customized with colors and `linetype` arguments.

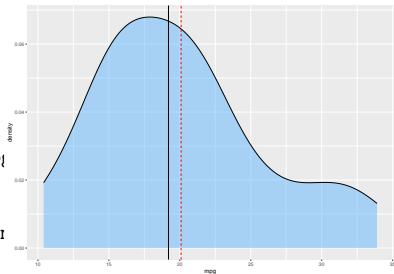
```
ggplot(mtcars,
       aes(x = mpg)) +
  geom_density(fill="steelblue1",
              alpha=0.5) +
  geom_vline(xintercept = mean(mtcars$mpg),
            color="red",
            linetype="dashed")
```



More on ggplot: density plots

- You can have multiple geometries at once.

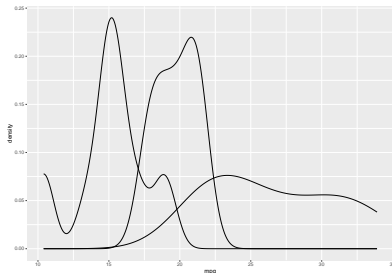
```
ggplot(mtcars,
  aes(x = mpg)) +
  geom_density(fill="steelblue1",
    alpha=0.5) +
  geom_vline(xintercept = mean(mtcars$mpg),
    color="red",
    linetype="dashed") +
  geom_vline(xintercept = median(mtcars$mpg),
    color="black",
    linetype="solid")
```



More on ggplot: density plots

- In the aesthetics, you can include an index factor to the `group=` argument to subset a geometry in different levels

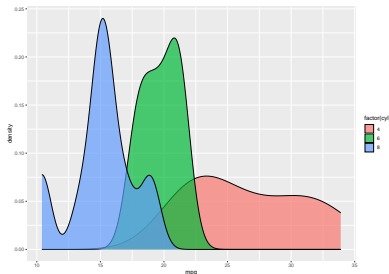
```
ggplot(mtcars,  
  aes(x = mpg,  
      group=factor(cyl))) +  
  geom_density()
```



More on ggplot: density plots

- When doing so, make sure to provide color to all the factor levels by setting fill or color arguments in the general aesthetics `aes()` argument.

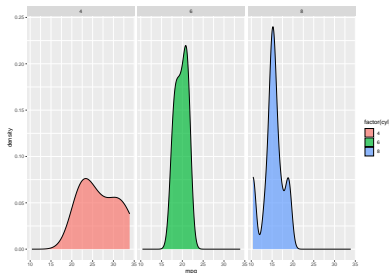
```
ggplot(mtcars,  
  aes(x = mpg,  
      group=factor(cyl),  
      fill=factor(cyl))) +  
  geom_density(alpha=0.7)
```



More on ggplot: facets

- ▶ When a group factor has many levels, is sometimes useful to split the plot in multiple facets using the geometries of `facet_wrap` or `facet_grid`.
 - ▶ Notice the syntax, requires a `~` before the factor.

```
ggplot(mtcars,  
  aes(x = mpg,  
      group=factor(cyl),  
      fill=factor(cyl))) +  
  geom_density(alpha=0.7) +  
  facet_wrap(~ factor(cyl))
```



Grammar of graphics in ggplot2

- ▶ Browse the most common [named colors in R](#).
- ▶ ggplot2 is a powerful tool for creating professional visualizations.
- ▶ Search on the internet or ask ChatGPT for help with specific plot types using keywords based on geometries, such as **line plots**, **histograms**, **boxplots**, **coefficient plots**, etc.

reshaping data with dplyr

- ▶ **Reshaping** a data frame is a crucial skill in data science that enables you to perform various necessary tasks efficiently.
- ▶ There are two main types of data structures: **long** and **wide** formats.
- ▶ **Long format** is the preferred structure for most R functions and packages, including ggplot2. It is **tidy data** that is easy to manipulate and analyze.
- ▶ Although not tidy, **wide format** can be useful for presenting tables to audiences as it conveys more information in a smaller space. However, it is not ideal for data analysis.

long vs wide

country	year	metric
x	1960	10
x	1970	13
x	2010	15
y	1960	20
y	1970	23
y	2010	25
z	1960	30
z	1970	33
z	2010	35

`pivot_wider(names_from = "year",
names_prefix = "yr",
values_from = "metric")`

country	yr1960	yr1970	yr2010
x	10	13	15
y	20	23	25
z	30	33	35

`pivot_longer(cols = yr1960:yr2010,
names_to = "year",
names_prefix = "yr",
values_to = "metric")`

reshaping data with dplyr

- ▶ The `pivot_` functions allow you to **reshape** data frames from long to wide or vice versa, which can be useful for data wrangling and visualization purposes.

```
# Create example data
(df <- data.frame(
  id = c(1, 2, 3),
  treatment = c("A", "B", "C"),
  day1 = c(10, 15, 12),
  day2 = c(12, 16, 18),
  day3 = c(8, 14, 10)
))
```

```
##   id treatment day1 day2 day3
## 1  1         A   10  12    8
## 2  2         B   15  16   14
## 3  3         C   12  18   10
```

reshaping data with dplyr

- ▶ `pivot_longer()` is used to convert a wide data frame to a **long format** by stacking columns into rows.
- ▶ You must specify which **columns** to pivot, the **names** for the new columns, and the name of the column to store the **values**.

```
# Use pivot_longer() to reshape  
# the data from wide to long
```

```
df_long <-  
  pivot_longer(df,  
    cols = starts_with("day"),  
    names_to = "day",  
    values_to = "result")  
df_long
```

```
## # A tibble: 9 x 4  
##       id treatment day    result  
##   <dbl> <chr>    <chr>  <dbl>  
## 1     1 A      day1     10  
## 2     1 A      day2     12  
## 3     1 A      day3      8  
## 4     2 B      day1     15  
## 5     2 B      day2     16  
## 6     2 B      day3     14  
## 7     3 C      day1     12  
## 8     3 C      day2     18  
## 9     3 C      day3     10
```


reshaping data with dplyr

- ▶ while `pivot_wider()` does the opposite by spreading rows into columns.

`pivot_wider()` takes similar arguments, but also requires specifying which column to use for the column names and which column to use for the values in the new columns.

```
# Use pivot_wider() to reshape  
# the data from long to wide  
  
df_wide <-  
  pivot_wider(df_long,  
              names_from = "day",  
              values_from = "result")  
  
df_wide
```

```
## # A tibble: 3 x 5  
##       id treatment  day1  day2  day3  
##   <dbl> <chr>    <dbl> <dbl> <dbl>  
## 1     1 A      10    12     8  
## 2     2 B      15    16    14  
## 3     3 C      12    18    10
```

Merging datasets

- Merging two datasets is actually more complicated than you might think

a

x1	x2
A	1
B	2
C	3

+

b

x1	x3
A	T
B	F
D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::left_join(a, b, by = "x1")
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::right_join(a, b, by = "x1")
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::inner_join(a, b, by = "x1")
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::full_join(a, b, by = "x1")
Join data. Retain all values, all rows.

Merging datasets: key variables

```
polity <- read_csv("data/polity2.csv")  
gapminder <- read_csv("data/gapminder2.csv")  
names(gapminder)
```

```
## [1] "cntry"      "continent" "year"      "lifeExp"   "pop"      "gdpPercap"
```

```
names(polity)
```

```
## [1] "country" "year"    "polity"
```

```
# with dplyr, rename()  
gapminder <- gapminder %>% rename(country=cntry)
```

```
# with base R, either colnames() or names()  
colnames(gapminder)[1] <- "country"
```

```
names(gapminder)
```

```
## [1] "country"    "continent" "year"      "lifeExp"   "pop"      "gdpPercap"
```

Merging datasets: base R

```
dim(gapminder)
```

```
## [1] 1704      6
```

```
dim(polity)
```

```
## [1] 17228      3
```

```
merged_df <- merge(gapminder, polity,  
                   by = c("country", "year"))
```

```
nrow(merged_df)
```

```
## [1] 1324
```

Merging datasets: dplyr

```
merged_data1 <- inner_join(gapminder, polity)
merged_data2 <- full_join(gapminder, polity)
merged_data3 <- left_join(gapminder, polity)
merged_data4 <- right_join(gapminder, polity)
```

```
nrow(merged_data1)
```

```
## [1] 1324
```

```
nrow(merged_data2)
```

```
## [1] 17608
```

```
nrow(merged_data3)
```

```
## [1] 1704
```

```
nrow(merged_data4)
```

```
## [1] 17228
```