LEARNING THE TIDYVERSE

AN ILLUSTRATED GUIDE

BY COLLEEN O'BRIANT

CHAPTER 1: INTRODUCTON

CHAPTER 2: TIDIED DATA

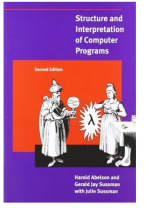
CHAPTER 3: DPLYR

CHAPTER 4: GGPLOT2

CHAPTER 5: PURRR

1 INTRODUCTON

THERE'S ONLY ONE CORRECT WAY TO BEGIN A BOOK ON PROGRAMMING, SO I'LL STEAL THE INTRO FROM HERE:



We are about to study the idea of a *computational process*. Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called *data*. The evolution of a process is directed by a pattern of rules called a *program*. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.

A computational process is indeed much like a sorcerer's idea of a spirit. It cannot be seen or touched. It is not composed of matter at all. However, it is very real. It can perform intellectual work. It can answer questions. It can affect the world by disbursing money at a bank or by controlling a robot arm in a factory. The programs we use to conjure processes are like a sorcerer's spells. They are carefully composed from symbolic expressions in arcane and esoteric *programming languages* that prescribe the tasks we want our processes to perform.

A computational process, in a correctly working computer, executes programs precisely and accurately. Thus, like the sorcerer's apprentice, novice programmers must learn to understand and to anticipate the consequences of their conjuring. Even small errors (usually called *bugs* or *glitches*) in programs can have complex and unanticipated consequences.

Fortunately, learning to program is considerably less dangerous than learning sorcery, because the spirits we deal with are conveniently contained in a secure way. Real-world programming, however, requires care, expertise, and wisdom. A small bug in a computer-aided design program, for example, can lead to the catastrophic collapse of an airplane or a dam or the self-destruction of an industrial robot.

Master software engineers have the ability to organize programs so that they can be reasonably sure that the resulting processes will perform the tasks intended. They can visualize the behavior of their systems in advance. They know how to structure programs so that unanticipated problems do not lead to catastrophic consequences, and when problems do arise, they can *debug* their programs. Well-designed computational systems, like well-designed automobiles or nuclear reactors, are designed in a modular manner, so that the parts can be constructed, replaced, and debugged separately.

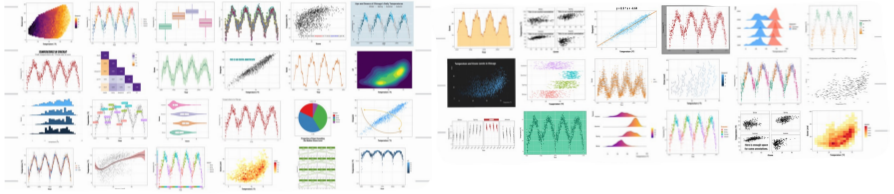
R'S BENEFITS

THERE ARE TONS OF LANGUAGES WE COULD BE LEARNING. WHY R?

1. R'S SYNTAX IS SIMPLE AND IF YOU APPROACH IT CORRECTLY, YOU CAN EASILY GO FROM NO PROGRAMMING SKILLS TO DOING SOPHISTICATED STUFF REALLY FAST.

This is one great advantage of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. All of the formal properties can be covered in an hour, like the rules of chess. After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues—figuring out what we want to compute, how we will decompose problems into manageable parts, and how we will work on the parts.

2. GGPLOT2 MADE THE TIDYVERSE AN INDUSTRY STANDARD FOR DATA ANALYTICS: NOTHING ELSE COMES CLOSE



3. ALL PROGRAMMING LANGUAGES ARE BASICALLY THE SAME, SO JUST LIKE THE ADVICE "THE BEST DIET IS THE ONE YOU STICK WITH", THE BEST PROGRAMMING LANGUAGE IS THE ONE THAT'S SIMPLE AND EASY ENOUGH TO MASTER. ONCE YOU LEARN ONE LANGUAGE WELL, LEARNING OTHERS IS 1000 TIMES EASIER.

4. AND FINALLY, USING R IS JUST GREAT FUN. I HOPE YOU'LL AGREE: IT'S EASY TO USE R TO CREATE ELEGANT + BEAUTIFUL SOLUTIONS TO ALL KINDS OF COMPLICATED DATA ANALYTICS PROBLEMS. AND THAT'S POWERFUL AND WONDERFUL.

R'S SHORTCOMINGS

R IS A VERY POWERFUL DATA ANALYTICS ENGINE.
BUT YOU MIGHT HEAR PEOPLE SAY THINGS LIKE "R IS SLOW".
SO WHAT'S GOING ON? IS IT A GOOD LANGUAGE OR NOT?

HERE'S WHAT I THINK: BEFORE YOU START WRITING A PROGRAM, YOU HAVE A VAGUE IDEA OF HOW YOU'D LIKE TO APPROACH SOLVING THE PROBLEM. AS YOU'RE WRITING IT, YOU'RE FORCED TO FORMALIZE THOSE IDEAS AND MAKE EACH STEP EXPLICIT. BY THE TIME YOU'VE FINISHED THE PROGRAM, YOU UNDERSTAND THE PROBLEM AT ANOTHER LEVEL. THIS IS THE INTELLECTUAL TASK OF PROGRAMMING, AND R IS AT LEAST AS GOOD AS ANY OTHER LANGUAGE AT HELPING YOU DO THIS PART.

BUT SOMETIMES YOU MIGHT HAVE SCALABILITY PROBLEMS: R STARTS SLOWING DOWN OR CRASHING (THIS WON'T BE AN ISSUE IN THIS COURSE OR EVEN MAJOR, BUT YOU MIGHT SEE IT AT YOUR FUTURE JOB). FOR INSTANCE, R DOES GREAT WITH DATASETS UP TO ABOUT A COUPLE MILLION ROWS. IF YOU NEED TO SCALE TO DATASETS WITH A COUPLE BILLION ROWS, YOU CAN RE-IMPLEMENT YOUR SOLUTION IN RUST / JAVA/C. IF YOU NEED TO SCALE TO DATASETS WITH A COUPLE TRILLION ROWS, YOU CAN USE SPARK OR FLINK TO DISTRIBUTE THE PROCESS AMONG SEVERAL COMPUTERS.

R IS A GREAT CHOICE FOR WRITING THAT FIRST PROGRAM THAT DOES THE HARD INTELLECTUAL TASK OF SOLVING THE PROBLEM AT HAND. AFTER YOU'VE DONE THAT, IF YOU FIND R IS TOO SLOW FOR WHAT YOU NEED TO DO, YOU CAN JUST RE-IMPLEMENT YOUR SOLUTION WITH THE BEST TECHNOLOGY FOR THAT USE CASE.

GOOD CODE IS ...

1. CODE THAT GETS YOU THE RIGHT ANSWER!

IF YOU GET THE RIGHT ANSWER, YOU CAN ALWAYS JUST LEAVE IT THERE.

IF YOU HAVE ERRORS, IT'S NOT READY TO TURN IN. KEEP WORKING AND COME TO OFFICE HOURS.

```
Error: unexpected ')'
```

2. CODE THAT SOLVES THE PROBLEM IN THE SIMPLEST POSSIBLE WAY.

LEARN ABOUT FUNCTIONS BY READING THEIR HELP DOCS.

```
> ?qelp::library
> ?qelp::install.packages
```

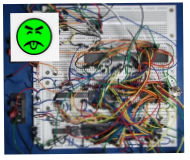
NOT BY ASKING THESE RESOURCES. THEY MAY HELP YOU GET THE ANSWER, BUT THEY WON'T HELP YOU DEVELOP YOUR UNDERSTANDING.

stackoverflow Google ChatGPT

3. CODE THAT IS CLEAR AND READABLE FOR OTHERS

"Programs must be written for people to read, and only incidentally for machines to execute."

— Harold Abelson, Structure and Interpretation of Computer Programs



4. CODE THAT INCLUDES COMMENTS THAT EXPLAIN WHY YOU'RE DOING WHAT YOU'RE DOING.

ANSWER "WHY", NOT "WHAT".

90% of all code comments:

2

TIDIED DATA

2.1 Vectors

IN R, DATA IS HELD IN VECTORS. YOU CAN CONSTRUCT A VECTOR USING THE FUNCTION `c()`. C IS SHORT FOR "COMBINE": YOU CAN COMBINE ELEMENTS TO FORM A VECTOR.

EXAMPLES:

1. COMBINE NUMBERS TO FORM A NUMERIC VECTOR:

```
> c(5, 3, 9)
[1] 5 3 9
```

2. COMBINE CHARACTER STRINGS TO FORM A CHARACTER VECTOR:

```
> c("apple", "banana", "strawberry")
[1] "apple"      "banana"     "strawberry"
```

NOTE: CHARACTER STRINGS NEED QUOTES AROUND THEM. NUMBERS SHOULD NOT HAVE QUOTES AROUND THEM.

YOUR TURN! COMPLETE `KOI_vectors.R` NOW. YOU'LL LEARN:

- MATH OPERATORS WORK ON VECTORS: `+ - / * ^`
- `min()` AND `sum()` WORK ON VECTORS
- USE `length()` TO FIND THE NUMBER OF ELEMENTS IN A VECTOR
- REPEAT ELEMENTS USING `rep()`
- DO RANDOM SAMPLING FROM A VECTOR USING `sample()`

2.2 Pipes

THE PIPE: `%>%` IS THE MOST FREQUENTLY USED FUNCTION IN THE TIDYVERSE.

HERE'S WHAT IT DOES:

SUPPOSE YOU HAVE SOME DATA `x` AND YOU'D LIKE TO APPLY SOME FUNCTION `f` ON IT. SO YOU RUN `f(x)`.

FOR EXAMPLE, TAKE THE VECTOR 1:3 AND FIND ITS MINIMUM BY APPLYING `min()`:

```
> min(1:3)
[1] 1
```

ANOTHER WAY TO DO THE SAME THING IS TO USE A PIPE:

```
x %>% f()
```

```
> 1:3 %>% min()
[1] 1
```

THE PIPE SIMPLY TAKES THE DATA THAT COMES BEFORE IT AND INSERTS IT → INTO THE FUNCTION THAT COMES AFTER.

THE WAY YOU SHOULD READ THE PIPE IS WITH THE WORD "THEN", AS IN: "TAKE X, THEN APPLY F."

THERE'S NO REASON TO STOP THERE: WHAT IF WE WANTED TO TAKE X, THEN APPLY F, THEN APPLY G, THEN APPLY H? USING PIPES:

```
x %>% f() %>% g() %>% h()
```

OR USING MULTIPLE LINES, THE PIPE MUST GO AT THE END OF THE LINE:

```
x %>%
  f() %>%
  g() %>%
  h()
```

VERSUS WITHOUT USING PIPES, YOU'D HAVE TO READ INSIDE-OUT.

```
h(g(f(x)))
```

ALL 3 OF THESE WILL OUTPUT THE SAME THING.

YOUR TURN! COMPLETE `K02_pipe.R` NOW. YOU'LL GET PRACTICE PIPING DATA THROUGH MULTIPLE FUNCTIONS AND COMBINING THE PIPE WITH A PERIOD `.` TO PIPE DATA INTO THE SECOND ARGUMENT OF A FUNCTION INSTEAD OF THE FIRST.

2.3 tibbles

TIBBLES ARE TIDYVERSE SPREADSHEETS. DATA IS STILL BEING HELD IN VECTORS (COLUMN VECTORS SPECIFICALLY), BUT THE ROWS OF A TIBBLE ALSO HOLD MEANING. THE ROWS ARE THE OBSERVATIONS WHILE THE COLUMNS ARE THE VARIABLES. THIS IS TRICKY TO UNDERSTAND, SO LET'S DO AN EXAMPLE:

DAILY WEATHER: LET'S WRITE DOWN EACH DAY'S HIGH TEMP, LOW TEMP, AND RAINFALL. OUR DATA IN WORDS:

- ON JAN 1, 2023 WE HAD A HIGH OF 46°, A LOW OF 37°, AND 0.07 INCHES OF RAIN.
- ON JAN 2, 2023, WE HAD A HIGH OF 46°, A LOW OF 35°, AND 0.00 INCHES OF RAIN.
- ON JAN 3, 2023, WE HAD A HIGH OF 47°, A LOW OF 34°, AND 0.08 INCHES OF RAIN.
- ETC

WHAT SHOULD THE OBSERVATIONS (ROWS) BE?
EACH DAY WE WENT OUTSIDE AND OBSERVED THE WEATHER.
SO EACH DAY SHOULD HAVE ITS OWN ROW.
WHAT ARE THE VARIABLES (COLUMNS) WE OBSERVED?
1) THE DATE, 2) HIGH TEMP, 3) LOW TEMP, AND 4) RAINFALL.

SO WE WANT OUR TIBBLE TO LOOK LIKE THIS:

DATE	HIGH TEMP	LOWTEMP	RAINFALL
1/1/23	46	37	0.07
1/2/23	46	35	0.00
1/3/23	47	34	0.08

OBSERVATIONS AS ROWS

VARIABLES AS COLUMNS

THE MANTRA "OBSERVATIONS AS ROWS, VARIABLES AS COLUMNS" IS WHAT WE CALL THE TIDIED DATA FORMAT. THERE ARE TONS OF WAYS YOU COULD FORMAT YOUR DATA, BUT THE TIDYVERSE IS COMPATIBLE WITH ONLY THIS WAY. LUCKILY, IT TURNS OUT TO BE VERY EXPRESSIBLE.

HERE'S THE CODE TO CONSTRUCT OUR TIBBLE:

```
> tibble(
+   date = as.Date(c("2023-01-01", "2023-01-02", "2023-01-03")),
+   high_temp = c(46, 46, 47),
+   low_temp = c(37, 35, 34),
+   rainfall = c(0.07, 0.00, 0.08)
+ )
# A tibble: 3 × 4
  date       high_temp low_temp rainfall
  <date>         <dbl>    <dbl>    <dbl>
1 2023-01-01        46       37     0.07
2 2023-01-02        46       35     0
3 2023-01-03        47       34     0.08
```

1. USE THE FUNCTION tibble()
2. tibble() TAKES A LIST OF VECTORS CREATED WITH c() THAT BECOME VARIABLE COLUMNS
3. EACH VARIABLE COLUMN HAS A NAME (date, high_temp, low_temp, rainfall)

2 RULES FOR TIBBLES:

1. EACH COLUMN MUST BE NAMED (date, high_temp, low_temp, rainfall ARE THE VARIABLE NAMES HERE). IF YOU TRY TO DEFINE A COLUMN WITHOUT GIVING IT A NAME, tibble() GENERATES ONE FOR YOU. (try this! take the code above and delete 'date =', 'high_temp =', etc.)

2. EACH COLUMN MUST HAVE THE SAME NUMBER OF ROWS. IF YOU TRY TO DEFINE A COLUMN THAT'S SHORTER THAN THE OTHERS, tibble() WILL THROW AN ERROR. (EXCEPTION: IF YOU DEFINE A COLUMN WITH ONLY ONE ELEMENT, tibble() WILL REPEAT IT TO MAKE IT THE SAME LENGTH AS THE OTHER COLUMNS).

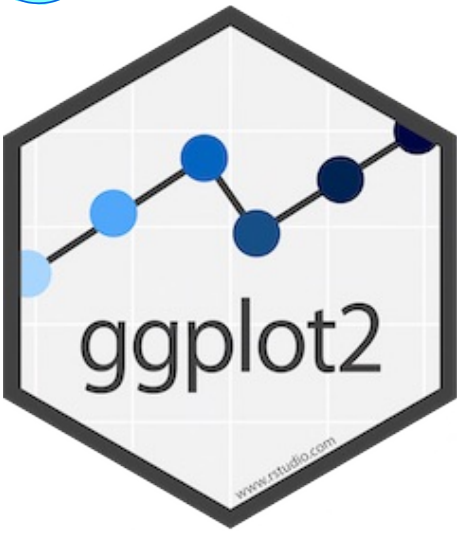
YOUR TURN! COMPLETE `K03_tibble.R` NOW. YOU'LL PRACTICE CONSTRUCTING YOUR OWN TIBBLE AND THEN YOU'LL LEARN HOW TO:

- `<-` ASSIGN IT TO A VARIABLE NAME IN YOUR ENVIRONMENT
- `view()` IT IN A SEPARATE TAB
- FIND ITS DIMENSIONS USING `nrow()` AND `ncol()`
- FIND ITS VARIABLE NAMES USING `names()`
- ADD A NEW OBSERVATION USING `add_row()`

COMING SOON...

3 DPLYR

4 GGPLOT2



4.1 intro · recipes for success

4.2 intermediate: grammar of graphics

4.3 advanced: animations and maps

4.1 intro · recipes for success

IN THIS CHAPTER, I'LL GIVE YOU A HANDFUL OF RECIPES FOR DRAWING GGPLOTS: I'LL SHOW YOU HOW TO BUILD A BAR PLOT, A HISTOGRAM, A BOX PLOT (AKA "BOX-AND-WHISKERS" PLOT), A SCATTERPLOT, AND HOW TO ADD A LINE OF BEST FIT TO THAT SCATTERPLOT.

BUT FIRST, WE NEED SOME DATA. AS YOU'LL SEE IN THIS CHAPTER, THE TYPE OF PLOT YOU'LL WANT TO DRAW DEPENDS CRUCIALLY ON THE TYPE OF DATA YOU HAVE. SUPPOSE WE HAVE A TIBBLE CALLED `students` THAT LOOKS LIKE THIS:

FIRST-SEMESTER MATH GRADE

FINAL GRADE IN THEIR MATH COURSE

sex	study_time	grade1	final_grade
"female"	5-10 H	94.9	95.4
"male"	2-5 H	79.6	73.7
"female"	5-10 H	64.2	49.1
...	...	...	...

HOURS PER WEEK STUDYING MATH

WE HAVE SOME VARIABLES WHICH ARE CATEGORICAL (`sex` TAKES ON EITHER "male" OR "female"; `study_time` TAKES ON "0-2H", "2-5 H", ETC). WE HAVE OTHER VARIABLES WHICH ARE NUMERIC, AND IN PARTICULAR, THEY ARE CONTINUOUS (`grade1` AND `final_grade` CAN TAKE ON ANY VALUE BETWEEN 0 AND 100).

THE ALTERNATIVE TO A CONTINUOUS NUMERIC IS A DISCRETE NUMERIC. CONTINUOUS VARIABLES ARE MEASURED WHILE DISCRETE VARIABLES ARE COUNTED.

CONTINUOUS: MEASURED LIKE HOW MUCH AN M&M WEIGHS

DISCRETE: COUNTED LIKE HOW MANY M&M'S YOU HAVE

WHEN YOU STUDY DUMMY VARIABLES, YOU'LL LEARN HOW TO ENCODE A CATEGORICAL VARIABLE WITH 0'S AND 1'S TO USE IT AS A DISCRETE VARIABLE. SO THE DISTINCTION BETWEEN CATEGORICAL VARIABLES AND DISCRETE VARIABLES IS ACTUALLY NO DISTINCTION AT ALL. THAT'S WHY, FOR THE REST OF THIS CHAPTER, I'LL REFER TO BOTH CATEGORICAL AND DISCRETE VARIABLES AS "DISCRETE".

SO `students` HAS 4 VARIABLES: THE FIRST TWO ARE DISCRETE AND THE SECOND TWO ARE CONTINUOUS.

Students

sex	study_time	grade1	final_grade
"female"	5-10 H	94.9	95.4
"male"	2-5 H	79.6	73.7
"female"	5-10 H	64.2	49.1
...	...	...	...

DISCRETE CONTINUOUS

ON TO THE RECIPES!

RECIPE 1: BAR PLOTS

geom_bar()

RECIPE

RECIPES 1 AND 2: BAR PLOTS AND HISTOGRAMS

USE THESE PLOTS FOR VISUALIZING THE DISTRIBUTION OF A SINGLE VARIABLE.

IF THE VARIABLE IS CATEGORICAL / DISCRETE, YOU SHOULD USE A BAR PLOT.

IF THE VARIABLE IS CONTINUOUS, YOU SHOULD USE A HISTOGRAM.

HOW MANY HOURS DO STUDENTS SPEND STUDYING?

→ DISTRIBUTION OF A SINGLE DISCRETE VARIABLE: USE A BAR PLOT

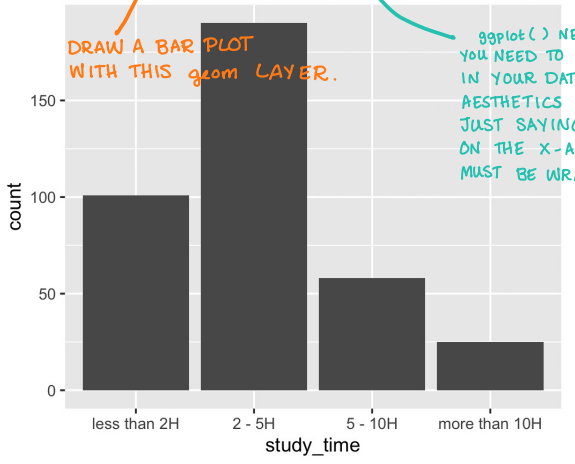
PIPE YOUR TIBBLE INTO THE FUNCTION ggplot()

```
students %>%
  ggplot(aes(x = study_time)) +
  geom_bar()
```

AFTER THE ggplot() CALL, LAYERS ARE ADDED WITH +.

DRAW A BAR PLOT WITH THIS geom LAYER.

ggplot() NEEDS AN AESTHETIC MAPPING: YOU NEED TO TELL IT WHICH VARIABLES IN YOUR DATASET MAP TO WHICH VISUAL AESTHETICS IN THE PLOT. HERE WE'RE JUST SAYING TO DRAW study_time ON THE X-AXIS. AESTHETIC MAPPINGS MUST BE WRAPPED IN A aes().



RECIPE 2 : HISTOGRAMS

geom_histogram( )

WHAT'S THE GRADE DISTRIBUTION?

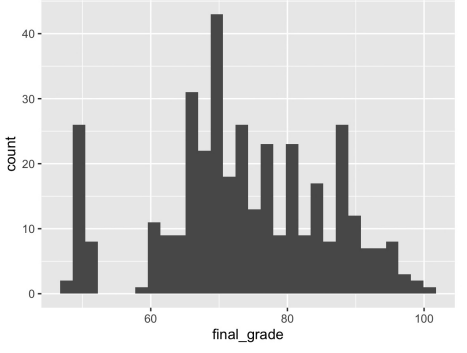
→ DISTRIBUTION OF A SINGLE CONTINUOUS VARIABLE : USE A HISTOGRAM

SAME AS RECIPE 1, BEGIN BY PIPING YOUR DATA INTO ggplot( ).

ggplot( ) NEEDS AN AESTHETIC MAPPING WRAPPED IN aes( ).

THE GEOM TO DRAW A HISTOGRAM : geom_histogram( ).

```
students %>%
  ggplot(aes(x = final_grade)) +
  geom_histogram()
```



RECIPE 3: BOXPLOTS

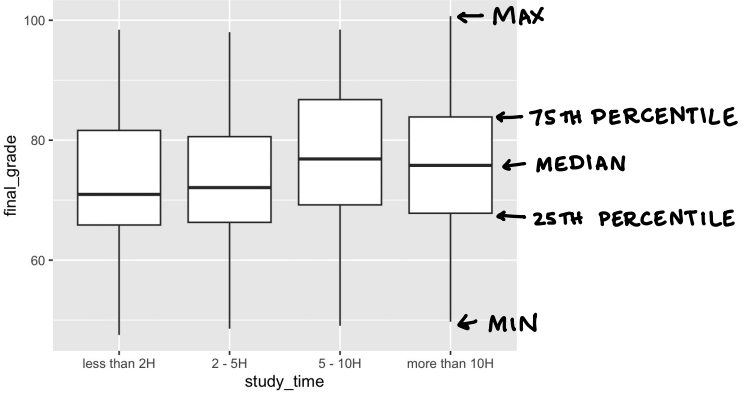
geom_boxplot()

RECIPE

RECIPE 3: USE A BOXPLOT TO VISUALIZE THE RELATIONSHIP BETWEEN ONE CATEGORICAL/DISCRETE VARIABLE AND ANOTHER CONTINUOUS VARIABLE.

DO STUDENTS WHO STUDY MORE (DISCRETE) EARN HIGHER GRADES (CONTINUOUS)?

```
students %>%
  ggplot(aes(x = study_time, y = final_grade)) +
  geom_boxplot()
```



BEGIN BY PIPING YOUR DATA INTO ggplot().

ggplot() NEEDS AN AESTHETIC MAPPING WRAPPED IN aes().

THIS TIME WE HAVE 2 VARIABLES: ONE FOR EACH AXIS.

THE GEOM TO DRAW A BOXPLOT: geom_boxplot().

RECIPE 4 : SCATTERPLOTS

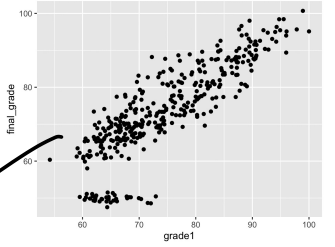
geom_point()

HOW WELL DOES A STUDENT'S FIRST-SEMESTER GRADE PREDICT THEIR FINAL GRADE IN A (HIGH SCHOOL) CLASS?

RECIPE

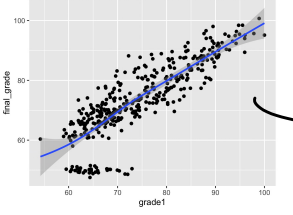
RECIPE 4 : USE A SCATTERPLOT TO VISUALIZE THE RELATIONSHIP BETWEEN TWO CONTINUOUS VARIABLES.

```
students %>%
  ggplot(aes(x = grade1, y = final_grade)) +
  geom_point()
```



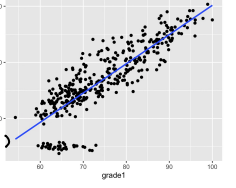
THIS LOOKS GOOD, BUT WHAT ABOUT ADDING + A LAYER ON TOP WITH A LINE OF BEST FIT: geom_smooth()

```
students %>%
  ggplot(aes(x = grade1, y = final_grade)) +
  geom_point() +
  geom_smooth()
```



WE CAN ALSO ASK geom_smooth TO USE A LINEAR MODEL (method = "lm") AND REMOVE THE PREDICTION STANDARD ERROR RIBBON (se = FALSE)

```
students %>%
  ggplot(aes(x = grade1, y = final_grade)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
```



RECIPE

RECIPE 5: WITH 2 DISCRETE VARIABLES, YOU CAN USE A BAR PLOT AND FACET BY OR FILL WITH THE SECOND VARIABLE.

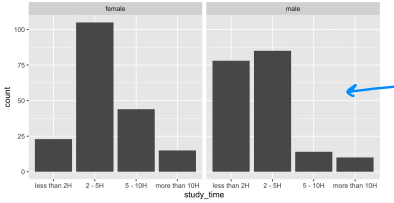
EXPLORING THE RELATIONSHIP BETWEEN 2 DISCRETE VARIABLES:

EX) DO FEMALES REPORT STUDYING FOR LONGER THAN MALES? (RELATIONSHIP BETWEEN `study_time` AND `sex`)

① TRY `geom_bar()` WITH `facet_wrap()` TO COMPARE THE BAR PLOT FOR FEMALES WITH THE BAR PLOT FOR MALES:

```
students %>%
  ggplot(aes(x = study_time)) +
  geom_bar() +
  facet_wrap(~ sex)
```

TILDE ~ INDICATES A FORMULA IS COMING: TO SEE THIS, REPLACE `sex` WITH `sex * romantic` TO SEE HOW BEING IN A ROMANTIC RELATIONSHIP EFFECTS MALE/FEMALE STUDY HOURS.

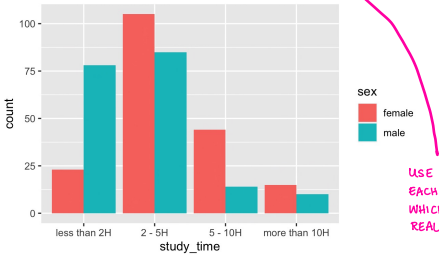


LOOKS LIKE FEMALES REPORT STUDYING MUCH MORE THAN MALES

② OR TRY A `fill` AESTHETIC MAPPING TO COLOR BARS SEPARATE COLORS FOR MALES VERSUS FEMALES:

```
students %>%
  ggplot(aes(x = study_time, fill = sex)) +
  geom_bar(position = "dodge")
```

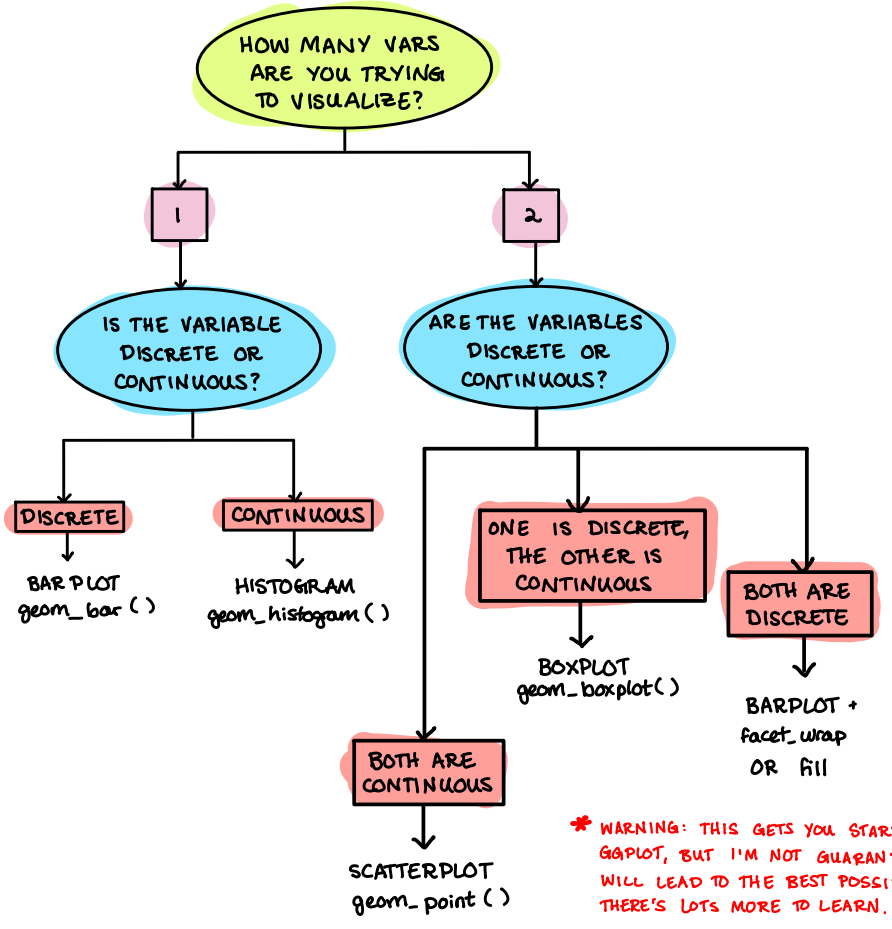
THIS IS AN AESTHETIC MAPPING JUST LIKE `x = study_time` BECAUSE IT TAKES A VARIABLE IN THE TIBBLE (sex) AND MAPS IT TO A PLOT AESTHETIC (fill).



USE `position = "dodge"` TO SET THE BARS NEXT TO EACH OTHER INSTEAD OF THE DEFAULT BEHAVIOR, WHICH IS TO STACK THEM (WHICH MAKES THE PLOT REALLY UNCLEAR).

GGPLOT BASICS: DECIDING BETWEEN .

geom_bar() geom_histogram() geom_point() geom_boxplot()



* WARNING: THIS GETS YOU STARTED ON USING GGPLOT, BUT I'M NOT GUARANTEEING THIS WILL LEAD TO THE BEST POSSIBLE PLOT! THERE'S LOTS MORE TO LEARN.

SUMMARY:

- AESTHETIC MAPPINGS GET WRAPPED IN `aes()` AND MAP VARIABLES IN YOUR TIBBLE TO AESTHETICS IN YOUR PLOT LIKE WHICH VAR GETS DRAWN ON THE X-AXIS, WHICH VAR GETS DRAWN ON THE Y-AXIS, AND WHICH VAR IS REPRESENTED BY COLOR.
- GEOMS ARE ADDED TO THE PLOT `+` AS LAYERS.

YOUR TURN! COMPLETE `K04_ggplot_intro.R` NOW. YOU'LL PRACTICE AESTHETIC MAPPINGS AND ALL THE RECIPES YOU LEARNED IN THIS SECTION.

COMING SOON...

4.2 intermediate: grammar of graphics

4.3 advanced: animations and maps

PURRR