UCSB ECON 145 Autograder Manual

Riley Berman, Alex Zhao, & 2024 ECON 145 Summer Team

Last Updated: 2025-03-28

Contents

Preface

This manual outlines how to develop an autograder for ECON 145, "Data Wrangling for Economics," at the University of California, Santa Barbara. Many of the techniques presented here can be applied to *any R-based autograder*.

You can read the manual online or download the **pdf** using the toolbar's "download" icon. For the best reading experience, the online version is recommended. In the online version, code blocks are scrollable and include a copy button in the top-right corner. Code blocks may look a little wonky in the pdf version.

This manual may contain mistakes or sections that require updates or improvement. Use the toolbar's "edit" icon to open the GitHub repository and submit issues or pull requests.

Acknowledgements

This guidebook aims to reflect new conventions and up-to-date techniques used in ECON 145 autograders developed by the 2024 ECON 145 Summer Team of **Jack Keefer (Head TA), Alex Zhao, Riley Berman, Shreya Sinha, and Michal Snopek**. We are extremely grateful for the foundational work established by previous ECON 145 TAs.

Special thanks to Alex Zhao and his excellent `autograder_manual_2024.pdf`, which laid the groundwork for this manual.

The author would also like to thank Xiao Yang for her invaluable feedback and suggestions.

Riley Berman

1 Introduction

This manual explains how to build an autograder for UCSB ECON 145 coding assignments.

> **WHAT IS AN AUTOGRADER?**
>
> In this manual, an *autograder*, or *autograder program*, is an R script that grades a student's R-based assignment and provides question-specific feedback. It runs alongside a set of supporting R scripts, collectively known as the *autograder infrastructure*.
>
> See Autograder Conventions for more details.

In ECON 145 assignments, coding questions are classified as either **Public** or **Private**.

- For any Public Question, we write custom "Checks" – lines of code that test for specific answer attributes – that evaluate the student's answer and return *constructive and dynamic feedback*.
- For any Private Question, we use a set of premade functions that evaluate the student's answer and return *necessary and basic feedback*.

Accordingly, most of this manual focuses on writing Checks for Public Questions.

We will look at grading four types of data structures: tibbles, vectors, lists, and plots. Moreover, as the majority of ECON 145 assignment questions are tibble-based, we direct our attention to grading tibbles.

For fellow ECON 145 TAs, we recommend skimming through `00-manual1.pdf` and `how_to_create_an_autograder_2024.pdf` for an overview of the autograder infrastructure.

> **Note:**
>
> Unless noted, all references and examples come from the Fall 2024 ECON 145 course.

2 Autograder Conventions

The *autograder infrastructure*, or *autograder system*, is a set of R scripts that work together to implement the *autograder* – that is, to evaluate a student's R-assignment and return question-specific feedback through the Gradescope platform.

Sometimes, the term "autograder" is used to refer to the entire system, but I will try to stick to this division throughout the manual.

WHY FOCUS ON JUST THE AUTOGRADER?

TAs primarily work on writing the autograder, while the Head TA handles the integration of the autograder and its infastructure with Gradescope.

A quick overview of the DGP.R file will also be provided, which *every TA* should review.

For implementation and testing details of the autograder, refer to the document `how_to_create_an_autograder_2024.pdf`.

2.1 General Structure

Generally, an autograder script will look something like:

```
rm(list = ls())

#--------------Set This------#
#loc       <- "local" # either "local", or "gradescope"
loc       <- "gradescope"

#------DON'T TOUCH THIS------#

#Setting working directory to source file location
if(loc=="local"){
  setwd(dirname(rstudioapi::getSourceEditorContext()$path))
}
source("inputs.R")
source(paste0(here::here(),"/helper_functions/autograder_setup.R",""))
source(paste0(here::here(),"/helper_functions/misc_helper_functions.R",""))
```

```
source("inputs.R")

#----------------------------#

if(status!="Error"){

  #Answer Key Goes Here...

  #Question 1 Solution

  #Question 2 Solution, etc.

  #Autograder Code Goes Here...

  #Testing Student's Question 1 Against Question 1 Solution

  #Testing Student's Question 2 Against Question 2 Solution, etc.

  # ------------------------------------------------------------ #

  JSONmaker(test.results, loc)
}
```

Under `#Answer Key Goes Here...`, TAs insert the assignment solutions. To distinguish solutions from the student's answers (so they can be compared), we tend to follow the convention of appending "`_test`" to the answer key's variable name.

- For example, if Question 1 on the homework asks the student to create a tibble named `basketball_data`, the answer key's corresponding tibble will be named `basketball_data_test`.

Under `#Autograder Code Goes Here...`, TAs insert their code for the autograder. Question by question, this code will compare the student's solution to the corresponding answer key. As a reminder...

- For any Public Question, we program a series of "Checks" (lines of code) to test the student's answer for a *range of attributes* and return dynamic feedback. See Public Questions.
- For any Private Question, we employ a group of built-in functions to test the student's answer for a *fixed set of attributes* and return basic feedback. See Private Questions.

The autograder **only** evaluates the student's final result, the R object – it does not grade the code required to produce it.

Accordingly, students can receive full credit for a question even if their code is vastly different from the answer key. The feedback our Checks generate guide students towards the *recommended, class-based* solution.

The student's R object **does not** need to be identical to the answer key to receive full credit. See Correct Check for more details.

2.2 What is test.results?

The autograder infrastructure has provided a `test.results` data frame to store and display feedback from our Checks.

When writing autograder code for *any* question, we *always* first initialize the question's default `test.results`.

For example, to initialize Question 1's `test.results`, we write the following line:

```
#Testing Student's Question 1 Against Question 1 Solution
test.results[1, ] <- c("Part 1 Question 1 (Public)", 0, 20, "Try again.")
```

where each element of `test.results[#, ]` is as follows:

Element	Description
1	the question's number
"Part 1 Question 1 (Public)"	the question's displayed part, number and type (Private/Public)
0	the question's default score
20	the question's maximum score
"Try again."	the default feedback message

It is important to note that a question's *displayed number* may differ from its *question number*, especially in the case of multi-part questions.

Recommendation: If students would greatly benefit from a very specific hint (e.g., using a certain function), you can include it in the default feedback message to ensure they see it in the case that none of the Checks trigger.

After initializing a question's `test.results[#, ]`, we can update it through indexing.

- For example, if the student's Question 1 is correct, we can update `test.results[1, ]` to award them full score with the following line:

```
#Modifying the score for Question 1
test.results[1, 2] <- 20
```

Similarly, we can modify the feedback message through the following line:

```
#Modifying the feedback message for Question 1
test.results[1, 4] <- "Insert feedback message here!"
```

Note:

For most of the sample code provided for Public Questions, we will be updating `test.results[2, ]` (i.e., Question 2).

3 Public Questions

Public Question Checks are designed to give students *constructive and dynamic feedback* that help them to improve their answer and progress toward the solution.

To achieve this, Public Questions will require multiple Checks that test for particular answer properties and generate tailored feedback.

There are two categories of Checks: **General** and **Special Checks**.

WHAT IS A GENERAL CHECK?

A *General Check* tests for a *common* answer property that applies across many, if not all, questions.

For example, all questions require the student to label their variable *a specific name*. Accordingly, for any Public (or Private) Question, a General Check should alert the student if their variable name does not match the name specified in the instructions (this is the Name Check).

We implement General Checks for all Public Questions.

WHAT IS A SPECIAL CHECK?

A *Special Check* tests for an answer property *unique* to the given question.

For example, if a Public Question asks the student to multiply a column by a certain number, a Special Check should alert them if they use a specific, improper scale (this is the Calculation Check). Note that this Check is *specific to this question* and is not a general answer property.

We implement Special Checks on a question-by-question basis.

Note:

General Checks are denoted with a "(G)" and Special Checks with an "(S)".

3.1 General Form

A Public Question's autograder code *combines* General and Special Checks like the following:

```
#Public Question 2 Autograder Code
#Testing Student's Question 2 Against Question 2 Solution

#Initializing test.results[2, ]
test.results[2, ] <- c("Part 1 Question 2 (Public)", 0, 20, "Try again.")

#General Check 1
if(test_condition){
  code...
#General Check 2
}else if(test_condition){
  code...
#Special Check 1
}else if(test_condition){
  code...
#Special Check 2
}else if(test_condition){
  code...
.
.
.
#Last General Check
}else if(test_condition){
  code...
}
```

We will learn a variety of Checks in the following sections.

3.2 Name Check (G)

Purpose: Checks whether the student's variable has the correct name.

Motivation: All questions require the student to give their variable a specific name. Without a Name Check, the autograder can't match and compare the student's answer to the answer key – and other Checks that rely on a correctly named student variable, such as the Column Name Check, may error out as a result.

```
#Name Check Example

if(is.error(variable_name)){
  test.results[2, 4] <- "`variable_name` is not found. Please make sure the variable is
 ↪  named correctly. (Any additional feedback as needed.)"
}
```

Technicals

Always include the Name Check.

If there is no Prerequisite Check, place this Check first (the `if` statement); otherwise, it should follow immediately after (the first `else if` statement).

Note:

The `is.error()` function comes from the `berryFunctions` package [Boessenkool, 2024]. For the ECON 145 autograder, this library should already be loaded in from the file `helper_functions/packages.R`.

3.3 Prerequisite Check (G)

Purpose: Checks whether the student got the prerequisite question correct.

Motivation: Many questions build on previous ones. If the student did not get the "prerequisite" question(s) correct, their current answer is likely incorrect as well. This Check prompts the student to revisit their earlier work. Since the prerequisite question is often the previous question, this Check is also called the "Previous Question Check."

```
#Prerequisite Check Example

else if(test.results[1, 2] == 0){ #If Question 1 is incorrect...
  test.results[2, 4] <- "This question depends on Question 1 being correct. Try again.
 ↪  (Any additional feedback as needed.)"
}
```

Technicals

Place the Prerequisite Check first so it is triggered first.

Don't use the Prerequisite Check for the first question, or for any stand-alone questions.

Recommendation: If the assignment has multiple parts (e.g., Part 1, Part 2, etc.), you should clarify where the prerequisite question is in the feedback message of `test.results[#, 4]`.

3.4 Structure Check (G)

Purpose: Checks whether the student's variable has the correct data structure (e.g., list, tibble). In most cases, this will be a tibble.

Motivation: Using the wrong data structure may cause the autograder to reject an otherwise correct-looking answer. It can also break Checks that expect a certain data structure, like the Calculation Check.

```
#Structure Check Example

#For a tibble...
else if(!is_tibble(variable_name)){
  test.results[2, 4] <- "Make sure `variable_name` is a tibble. (Any additional feedback
↪  as needed.)"
}

#For a list...
#`janitor::tabyl()` and other `janitor` related functions produce a list, not a tibble
#In this case, the Structure Check should check for a list, like below
else if(!is_list(variable_name)){
  test.results[2, 4] <- "Make sure `variable_name` is a list. (Any additional feedback
↪  as needed.)"
}
```

Technicals

Always include the Structure Check.

A common mistake students make is using `read.csv()` instead of `read_csv()` when loading datasets into R. The problem is that the former returns a *data frame*, while the latter returns a *tibble* (a special, modern type of data frame).

There are slight differences [Wickham and Grolemund, 2016] between how general data frames and tibbles behave in R. For example, tibbles do not change the type of the inputs (e.g., they never convert strings to factors) and they allow for non-syntactic names (e.g., names with spaces). These small differences may cause the autograder to reject the student's answer or inconvenience the student when solving the assignment.

For Public Questions that involve `janitor::tabyl()` and other `janitor` related functions (e.g., `adorn_pct_formatting`, `adorn_totals`), the student's answer will be a *list, not a tibble.* In this case, the Structure Check should check for a list. For three-way `tabyl` lists, see `private_tabyl_grader()`.

Recommendation: For questions that involve loading data into R, tack on a reminder in `test.results[#, 4]` for the student to use `read_csv()` instead of `read.csv()`.

3.5 Column Name Check (G)

Purpose: Dynamically checks whether the column names in the student's tibble (A) match those in the answer key (B). We break this test into two parts:

1. Test whether $B \not\subset A$, that is, if the student's tibble is missing any column names from the answer key.

2. Test whether $A \not\subset B$, that is, if the student's tibble has any additional column names not in the answer key.

Motivation: Many questions involve the deletion or addition of columns in a tibble. Students often misname, omit, or forget to remove such columns. This Check helps catch these issues early, reducing the time students spends sifting through their code for a simple mistake (e.g., a typo).

```
#Column Name Check Example

#First Test (B not in A)
else if(!all(colnames(variable_name_test) %in% colnames(variable_name))){
  test.results[2, 4] <- paste0(c("The following column(s) should be in `variable_name`,
↪  but they were not found in your answer: ",
    paste0(colnames(variable_name_test)[!(colnames(variable_name_test) %in%
    ↪  colnames(variable_name))], collapse = ", "), ". (Any additional feedback as
    ↪  needed.)"), collapse = "")
}

#Second Test (A not in B)
else if(!all(colnames(variable_name) %in% colnames(variable_name_test))){
  test.results[2, 4] <- paste0(c("The following column(s) should not be in
↪  `variable_name`, but they were found in your answer: ",
    paste0(colnames(variable_name)[!(colnames(variable_name) %in%
    ↪  colnames(variable_name_test))],
    collapse = ", "), ". (Any additional feedback as needed.)"), collapse = "")
}
```

Technicals

Always include the Column Name Check.

This Check is *essential* for subsequent Checks that rely on the student's answer having correctly named columns, like the Correct Check.

While the Column Name Check is designed for tibbles, it can easily be adapted for named lists by swapping `colnames()` for `names()`. Please note that lists should *first* be checked if they are named through a `NULL` Check like `is.null(names(variable_name))`.

3.6 Row & Column Check (G)

Purpose: Checks whether the student's tibble has the same number of rows and columns as the answer key.

Motivation: Many questions involve the deletion or addition of rows and/or columns in a tibble. This Check alerts students, rather broadly, to a discrepancy in their number of rows and columns. Row and column mismatches can cause Checks, like the Correct Check, to error out.

```
#Row & Column Check Example

#Row Check
else if(nrow(variable_name) != nrow(variable_name_test)){
  test.results[2, 4] <- "`variable_name` has the incorrect number of rows. (Any
 ↪  additional feedback as needed.)"
}

#Column Check
else if(ncol(variable_name) != ncol(variable_name_test)){
  test.results[2, 4] <- "`variable_name` has the incorrect number of columns. (Any
 ↪  additional feedback as needed.)"
}
```

Technicals

Always include the Row & Column Check.

Please note that the Column Check is *redundant* if it follows the Column Name Check. That is, if the student's answer passes the Column Name Check, it inevitably passes the Column Check. However, there are cases where the Column Check should *precede* the Column Name Check (e.g., if the tibble is very large, it is reasonable to first ensure that the number of columns is correct before checking for column names to potentially avoid a long Column Name Check message).

You may notice that 2024 ECON 145 autograders have this redundant Column Check. This is not really necessary, but is done to completely ensure that the Correct Check works smoothly.

The Row & Column Check can easily be adapted for vectors by swapping `nrow()` and `ncol()` for `length()`.

3.7 Correct Check (G)

Purpose: Checks whether the student's tibble and the answer key are *essentially* the same. If so, full points are awarded.

Motivation: This Check is the most important. It allows for some flexibility between the student's answer and the answer key, such as minor variations in column order, row order, rounding, and object attributes.

```
#Correct Check Example

else if(isTRUE(all.equal(variable_name |> ungroup() |>
                         select(colnames(variable_name_test)) |>
                         arrange(across(everything())),
                         variable_name_test |> ungroup() |>
                         select(colnames(variable_name_test)) |>
                         arrange(across(everything())),
                         tolerance = 0.001, check.attributes = F))){
  test.results[2, 2] <- 20 #Full credit
  test.results[2, 4] <- "Well done!"
}
```

Technicals

Always include the Correct Check.

Place this Check last (the last `else if` or the `else` statement).

This Check assumes that the student's column names match the answer key, which is why the Column Name Check is essential **beforehand**.[a]

[a] **Acknowledgment**: Previously, `all_equal()` was used for this Check. However, since `all_equal()` was deprecated in `dplyr 1.1.0`, we opted to use `all.equal()` instead. Unfortunately, because `all.equal()` does not contain the `ignore_col_order` and `ignore_row_order` arguments that allow for different column and row ordering, we had to implement this flexibility manually.

We set the `tolerance` to a small value (i.e., `0.001`) to allow for some flexibility in numeric rounding, because different orders of operation can produce slightly different numbers due to how floats are handled by the computer.

The argument `check.attributes = F` ignores the attributes (essentially the additional information attached to an R object) when comparing the student's answer to the answer key. In other words, we still want to award full credit to a student's answer that looks *virtually identical* to the solution, but contains internally *different metadata*.

- For example, the function `na.omit()` attaches an attribute to a tibble, making it internally different but visually identical to a tibble produced with `drop_na()`. Similarly, answers generated by the `tabyl()` function versus those created through a string of `mutate()` and `summarize()` functions.

The Correct Check can easily be adapted for named lists by swapping `colnames()` for `names()` and for vectors by implementing a reduced version of this Check.

3.8 PERMID Check (G)

Purpose: Checks whether the student provided a valid PERMID.

Motivation: This Check is *specific* to the ECON 145 autograder. In all assignments, students must provide their PERMID (a sequence of digits) at the top of their R script, which is used to randomly generate data (see DGP.R). This Check flags missing (the default PERMID is `1`) or generic (i.e., `1234`) student PERMIDs.

```
#PERMID Check Example

#Make sure students actually entered their PERMID----------------------------
#If the student does not input PERMID as instructed by the prompt
#The default PERMID will be 1
if(isTRUE(all.equal(PERMID, 1))){
  test.results[, 4] <- "Please follow the assignment prompt and input your PERMID!"
  test.results[, 2] <- 0
} else if (isTRUE(all.equal(PERMID, 1234))){
  test.results[, 4] <- "Please follow the assignment prompt and input your PERMID!"
  test.results[, 2] <- 0
}
```

Technicals

Always implement the PERMID Check once, at the bottom of the autograder script (below all the Public and Private Question Checks).

3.9 Type Check (S)

Purpose: Checks whether a column of the student's tibble has the correct data type (e.g., numeric, character).

Motivation: Some questions involve the conversion of a column from one data type to another. This Check detects an incorrectly converted (or unconverted) column and can hint at its correct conversion (e.g., using `as.numeric()`). This Check is especially useful for questions that rely on a column being numeric to perform computations.

```
#Type Check Example
```

```
#For a numeric column...
else if(!is.numeric(variable_name$column_name)){
  test.results[2, 4] <- "`column_name` is not numeric. (Any additional feedback as
 ↪  needed.)"
}

#For a character column...
else if(!is.character(variable_name$column_name)){
  test.results[2, 4] <- "`column_name` is not character string. (Any additional feedback
 ↪  as needed.)"
}
```

Technicals

Since the Type Check assumes that the student's answer has the correct column name, it is essential that the Column Name Check is placed **beforehand**.

For examples of how to check multiple columns simultaneously, see below.

```
#Dynamic Type Check Examples

#Example 1: Checking two columns, column_A (character) and column_B (numeric)
else if(typeof(variable_name$column_A) != typeof(variable_name_test$column_A) |
        typeof(variable_name$column_B) != typeof(variable_name_test$column_B)){

  typeof_check <- c(typeof(variable_name$column_A) !=
 ↪  typeof(variable_name_test$column_A),
                    typeof(variable_name$column_B) != typeof(variable_name_test$column_B))

  typeof_names <- paste0(paste0(c("`column_A` (correct: character)",
                                  "`column_B` (correct: numeric)")
                                [typeof_check], collapse = " "),
                                ". (Any additional feedback as needed.)")

  test.results[2, 4] <- paste0("The following column(s) have the incorrect data type: ",
 ↪  typeof_names, ".")
}

#Example 2 (Experimental): Checking every column in a tibble
```

```
else if(!all(sapply(variable_name, typeof)[order(names(sapply(variable_name_test,
↪  typeof)))] ==
            sapply(variable_name_test, typeof)[order(names(sapply(variable_name_test,
             ↪  typeof)))])){

  typeof_check <- c(sapply(variable_name, typeof)[order(names(sapply(variable_name_test,
↪  typeof))) ==
                    sapply(variable_name_test,
                     ↪  typeof)[order(names(sapply(variable_name_test, typeof)))])

  typeof_names <- c(sapply(variable_name_test,
↪  typeof)[order(names(sapply(variable_name_test, typeof)))] |> names())

  typeof_names_check <- paste0(typeof_names[!typeof_check], collapse = " ")

  typeof_correct_check <- paste0(sapply(typeof_names[!typeof_check], function(names)
↪  typeof(variable_name_test[[names]])), collapse = " ")

  test.results[2, 4] <- paste0("The following column(s) have the incorrect data type: ",
↪  typeof_names_check, ". They should be of type: ", typeof_correct_check, ". (Any
↪  additional feedback as needed.)")
}
```

3.10 Value Check (S)

Purpose: Checks whether the values in a column of the student's tibble match the answer key.

Motivation: Some questions involve the transformation of a column's values – for example, multiplying a column by a formula. This Check detects an incorrectly transformed (or unaltered) column and can hint at its proper transformation. The `sort()` function allows for flexible column ordering.

```
#Value Check Example

else if(!isTRUE(all.equal(variable_name$column_name |> sort(na.last = T),
                          variable_name_test$column_name |> sort(na.last = T),
                          tolerance = 0.001,
                          check.attributes = F))){
  test.results[2, 4] <- "The values of `column_name` are incorrect. (Any additional
↪  feedback as needed.)"
```

```
}
```

Technicals

Since the Value Check assumes that the student's answer has the correct column name, it is essential that the Column Name Check is placed **beforehand**.

The Value Check is a *general check* for a column's values. For more specialized checks, see the Calculation Check, the NA Check, or the Expression Check.

By default, `NA` values are removed from a vector when sorted. The `na.last = T` argument ensures that the `NA` values are placed last in the sorted vector but not removed.

If it is reasonable to check every column in the tibble (e.g., a very complicated public question with many intermediate steps), you could implement a Complete Value Check, as shown below. Note that loops in `R` do consume considerable computational resources – there are certainly other and/or better ways to do this, like through the `sapply()` function.

```
#Complete Value Check Example
#By construction below, this check should be implemented as the final check (the last
↪  `else if` or the `else` statement)

else {
  q2ccheck <- c()
      #Compares every column in the student's tibble with the corresponding answer key
      ↪  column...
      for (colname in colnames(variable_name)){
        if(!isTRUE(all.equal(variable_name[[colname]] |> sort(na.last = T),
                             variable_name_test[[colname]] |> sort(na.last = T),
                             tolerance = 0.001,
                             check.attributes = F))){
          q2ccheck <- append(q2ccheck, colname)
        }
      }

#Returns a list of incorrect columns...
  test.results[2, 4] <- paste0(c("The following columns are incorrect: ",
                                 paste0(q2ccheck, collapse = ", "),
                                 "(Any additional feedback as needed.)"), collapse = "")
```

```
}
```

For examples of how to check multiple columns simultaneously, see below.

```
#Dynamic Value Check Example

else if(any(!isTRUE(all.equal(variable_name$column_A |> sort(na.last = T),
                              variable_name_test$column_A |> sort(na.last = T),
                              tolerance = 0.001, check.attributes = F))),
            !isTRUE(all.equal(variable_name$column_B |> sort(na.last = T),
                              variable_name_test$column_B |> sort(na.last = T),
                              tolerance = 0.001, check.attributes = F))){

  value_check <- c(!isTRUE(all.equal(variable_name$column_A |> sort(na.last = T),
                                     variable_name_test$column_A |> sort(na.last = T),
                                     tolerance = 0.001, check.attributes = F)),
                   !isTRUE(all.equal(variable_name$column_B |> sort(na.last = T),
                                     variable_name_test$column_B |> sort(na.last = T),
                                     tolerance = 0.001, check.attributes = F)))

  value_names <- c("column_A", "column_B")

  test.results[2, 4] <- paste0(c("The following column(s) are incorrect: ",
↪  value_names[value_check], ". (Any additional feedback as needed.)"), collapse = " ")
}
```

3.11 Calculation Check (S)

Purpose: Checks whether the values in a *numeric column* of the student's tibble have been scaled correctly.

Motivation: Some questions involve the multiplying of a column by a certain factor or formula (e.g., `1000`, the `log()` function, Fahrenheit to Celsius conversion). This Check detects a *specific*, improper scale for a column (e.g., accidentally inverting a formula) and can hint at its correct transformation.

```
#Calculation Check Examples

#Example 1: Checking if a column is unscaled (it should have been scaled by 1000)
else if(isTRUE(all.equal((variable_name$column_name * 1000) |> sort(na.last = T),
```

```
                          variable_name_test$column_name |> sort(na.last = T),
                          tolerance = 0.001,
                          check.attributes = F))){
  test.results[2, 4] <- "The values of `column_name` are incorrect. Hint: Did you scale
  ↪  this column by 1000? (Any additional feedback as needed.)"
}

#Example 2: Checking if a column inversely applied a subtraction formula
else if(isTRUE(all.equal((variable_name$column_name * -1) |> sort(na.last = T),
                          variable_name_test$column_name |> sort(na.last = T),
                          tolerance = 0.001,
                          check.attributes = F))){
  test.results[2, 4] <- "The values of `column_name` are incorrect. Hint: Did you flip
  ↪  the subtraction formula when calculating this column? (Any additional feedback as
  ↪  needed.)"
}
```

Technicals

Since the Calculation Check assumes that the student's answer has the correct column name, it is essential that the Column Name Check is placed **beforehand**.

This Check also relies on the student's column being numeric, so the numeric Type Check should be placed **before** as well.

- For instance, in Example 1, the column must be numeric for this line `variable_name$column_name * 1000` to work.

This is a subset of the Value Check. That is, if the Calculation Check is triggered, so too will the Value Check (but not the other way around).

Thus, for the Calculation Check to be effective, it should be placed **before** the Value Check (if both are implemented). In tandem, the Calculation Check first alerts the student to the use of a *specific*, improper scale (e.g., an unscaled column), then the Value Check alerts them to whether they have used *any* improper scale. In this way, the feedback can be more personalized.

- For example, what if a column conversion has multiple steps? Using both Checks can allow us to target our feedback more effectively.

Alternatively, the Calculation Check can be implemented **within** the Value Check, as shown below.

```
#Calculation Check combined with Value Check Example (with example hints)

#Value Check
else if(!isTRUE(all.equal(variable_name$column_name |> sort(na.last = T),
 ↪  variable_name_test$column_name |> sort(na.last = T)))){
  #Calculation Check
  if(isTRUE(all.equal((variable_name$column_name / 100) |> sort(na.last = T),
                      variable_name_test$column_name |> sort(na.last = T), tolerance =
 ↪  0.001))){
    test.results[2, 4] <- "The values of `column_name` are incorrect. Hint: Did you
 ↪  convert `column_name` to percentage format?"
  }
  else{
    test.results[2, 4] <- "The values of `column_name` are incorrect. Hint: (a) Look at
 ↪  the function `lead()`. (b) Did you use the appropriate `.by` argument when
 ↪  constructing `column_name`?"
  }
}
```

Please note that the Calculation Check does not have to be implemented alongside the Value Check, however, its scope is *far more limited*.

3.12 NA Check (S)

Purpose: Checks whether the values in a column of the student's tibble contain `NA` values.

Motivation: Some questions involve the removal of `NA` values from an existing column (e.g., through `filter(!is.na(column_name))`) or a newly constructed column (e.g., through the argument `na.rm = T`). This Check detects the presence of `NA` values in a column and can hint at their correct removal.

```
#NA Check Examples

#Example 1
else if(any(is.na(variable_name$column_name))){
  test.results[2, 4] <- "The `column_name` has `NA` values. (Any additional feedback as
 ↪  needed.)"
}
```

```
#Example 2 (Alternative)
else if(sum(is.na(variable_name$column_name) > 0)){
  test.results[2, 4] <- "The `column_name` has `NA` values. (Any additional feedback as
 ↪  needed.)"
}
```

Technicals

Since the NA Check assumes that the student's answer has the correct column name, it is essential that the Column Name Check is placed **beforehand**.

This is a subset of the Value Check.

Like the Calculation Check, the NA Check should be placed **before** or **within** the Value Check and does not have to be implemented alongside it.

For examples of how to check multiple columns simultaneously, see below.

```
#Dynamic NA Check Example (with an example hint)
#Note: Use the all() function instead of any() in examples where some `NA` values might
 ↪  exist in a column, but not every value in the column should be `NA`.

else if(any(all(is.na(variable_name$column_A)) |
            all(is.na(variable_name$column_B)) |
            all(is.na(variable_name$column_C)))){

  wrong_cols <- c(all(is.na(variable_name$column_A)),
                  all(is.na(variable_name$column_B)),
                  all(is.na(variable_name$column_C)))

  checked_cols <- c("column_A", "column_B", "column_C")

  test.results[2, 4] <- paste0(c("The following column(s) have all NA values: ",
 ↪  checked_cols[wrong_cols], " Hint: Did you set na.rm = T?"), collapse = " ")
}
```

3.13 Expression Check (S)

Purpose: Checks whether the values in a *string column* of the student's tibble are correctly transformed.

Motivation: Many questions involve string manipulation, formatting, and concatenation. This Check detects a *specific*, wrongly transformed attribute of the string transformation and can hint at its correct modification. This Check is one of the more *common* Special Checks and can take on many forms.

```
#Expression Check Examples (with example hints)

#Example 1: Checking if a column has extracted the apostrophe "s" (i.e., 's) from its
↪  values
else if(sum(str_detect(variable_name$column_name, "'s"), na.rm = T) > 0){
  test.results[2, 4] <- "There are `'s` in `column_name`. Please remove all of the `'s`
↪  when modifying `column_name`."
}

#Example 2: Checking if a column has correctly removed all observations that are not a
↪  full name (i.e., names that only contain the first initial and last name, like "D.
↪  Smith")
else if(sum(str_detect(variable_name$column_name, "\\w{1}\\.\\s{1}\\w+$"), na.rm = T) >
↪  0){
  test.results[2, 4] <- "Make sure to remove all observations that are not a full name
↪  (e.g., D. Smith) from `column_name`. Hint: Try functions like str_replace() or
↪  grepl() with the appropriate regular expression."
}

#Example 3: Checking if a column has correctly rounded its values to the hundredths
↪  place
else if(!all(str_detect(variable_name$column_name, "\\.\\d{2}$"))){
  test.results[2, 4] <- "All values in `column_name` should be formatted to the second
↪  decimal place. Hint: Try using the round() function with the `digits` argument."
}

#Example 4: Checking if a column has correctly converted its year values to the
↪  corresponding decade (e.g., 1987 should be converted to 1980)
else if(sum(variable_name$column_name) %% 10 != 0){
  test.results[2, 4] <- "Make sure `column_name` only contains multiples of 10 (e.g.,
↪  1950, 1960, etc.). Hint: Convert the year values into multiples of 10. For example,
↪  1987 should be converted to 1980. Consider using the `%/%` (integer division)
↪  operator, but there are other possible solutions as well."
```

```
}
```

Technicals

Since the Expression Check assumes that the student's answer has the correct column name, it is essential that the Column Name Check is placed **beforehand**.

This is a subset of the Value Check.

Like the Calculation Check and the NA Check, the Expression Check should be placed **before** or **within** the Value Check and does not have to be implemented alongside it.

For examples of how to check multiple columns simultaneously, see below.

```
#Dynamic Expression Check (with an example hint)

#Checking if three columns (column_A, column_B, column_C) have converted their values to
 ↪  lowercase...
else if(any(str_detect(variable_name$column_A, "[A-Z]"),
            str_detect(variable_name$column_B, "[A-Z]"),
            str_detect(variable_name$column_C, "[A-Z]"))){

  q2_upper_check <- c(any(str_detect(variable_name$column_A, "[A-Z]")),
                      any(str_detect(variable_name$column_B, "[A-Z]")),
                      any(str_detect(variable_name$column_C, "[A-Z]")))

  q2_upper_name <- c("column_A", "column_B", "column_C")

  test.results[2, 4] <- paste0(c("The following column(s) contain observations
                                 that are not converted to lowercase:",
                                 q2_upper_name[q2_upper_check],
                                 "Hint: use str_to_lower() when necessary."),
                                 collapse = "  ")
}
```

3.14 Miscellaneous Checks (S)

The following are some highly specific Special Checks that fall outside the common categories. While they may not be practical, their structure could serve as a reference when cre-

ating your own custom Special Checks.

```
#Miscellaneous Check Examples (with example hints)

#Example 1 (From HW 6, Q3): Checking if the column `mocodes` has correctly split its
↪  observations into multiple columns (named `mocodes_1`, `mocodes_2`, etc.) that each
↪  contain at most only one 4-digit M.O. code (e.g., the entry "1049 1304 1000" should
↪  be split into "1049" in `mocodes_1`, "1304" in `mocodes_2`, and "1000" in
↪  `mocodes_3`). The `separate_wider_delim()` function would be optimal to achieve
↪  this.
#Column Check
else if(ncol(variable_name) != ncol(variable_name_test)){
  #Checks if each mocodes column has at most one M.O. code (4 digits long)
  #Checks for a mistaken `delim` argument
  if(any(str_detect(colnames(variable_name), "mocodes"))){
    mocodes_col <- variable_name |> select(contains("mocodes")) |> mutate_all(~
↪  str_detect(., "\\d.*\\d.*\\d.*\\d")) |> colSums(., na.rm = TRUE)
    if(sum(as.vector(mocodes_col), na.rm = TRUE) > 0){
      test.results[2, 4] <- "Each `mocodes` column should have at most one M.O. code.
↪  Hint: Look at the argument `delim` in `separate_wider_delim()`."
    }
  }
  else{
    test.results[2, 4] <- "`variable_name` has the incorrect number of columns. Hint:
↪  Look at the function `separate_wider_delim()`."
  }
}
#Column Name Check
else if(!all(colnames(variable_name_test) %in% colnames(variable_name))){
  #Checks for a mistaken `names_sep` argument
  if(any(!str_detect(colnames(variable_name)[str_detect(colnames(variable_name),
   ↪  "mocodes")], "_"))){
    test.results[2, 4] <- "Every `mocodes` column should have an `_` between `mocodes`
↪  and its corresponding number. Hint: Look at the argument `names_sep` in
↪  `separate_wider_delim().`"
  }
  else{
    test.results[2, 4] <- paste0(paste0(c("The following column(s) should be in
↪  `variable_name`, but they were not found in your answer:",
                                colnames(variable_name_test)[!(colnames(variable_name_test)
                                       ↪  %in% colnames(variable_name))]),
                                      collapse = "  "), ". Hint: Look at the function
                                      ↪  `separate_wider_delim().`")
```

```
  }
}

#Example 2 (From H14, Q4a): Dynamically checking if a column contains the correct values
↪  (there are 4 different values/rows):
else if(!isTRUE(all.equal(variable_name$column_name |> sort(na.last = T),
↪  variable_name_test$column_name |> sort(na.last = T)))){

  check_names <- variable_name$column_name

  correct_names <- variable_name_test$column_name

  correct_check <- sapply(check_names, function(names) !names %in% correct_names)

  correct_check_names <- paste0(correct_names[correct_check], collapse = " ")

  test.results[2, 4] <- paste0("The following values are missing in `column_name`: ",
↪  correct_check_names, ". Hint: (a) Look at the function `case_when()`. (b) Make sure
↪  you converted the values correctly according to the prompt.")
}

#Example 3 (From Final, P1 Q1b): Dynamically checking if the columns of a tibble were
↪  correctly cleaned by removing the punctuation symbols `‡`, `-`, `†`.
else if(any(sapply(c('‡', '-', '†'), function(value) {any(variable_name |>
↪  summarise_all(~ value %in% .))}))){

  value_check <- sapply(c('‡', '-', '†'), function(value) {any(variable_name |>
↪  summarise_all(~ value %in% .))})

  value_names <- paste0(c('‡', '-', '†')[value_check], collapse = " ")

  test.results[2, 4] <- paste0("The following special characters were found in
↪  `variable_name`: ", value_names, ". Please clean the data before proceeding.",
↪  collapse = "  ")
}
```

3.15 Dynamic Checks

You may have noticed a pattern in our Dynamic Checks (i.e, checks that evaluate multiple test conditions simultaneously).

Below is a template for constructing a two-condition Dynamic Check.

Dynamic Checks for more than two test conditions follow similarly. The `sapply` function can be useful for evaluating *many* test conditions.

```
#Dynamic Check General Example (for two test conditions)

#Checking if either of the test conditions are true...
else if(any(c(test_condition1, test_condition2)){

  #Creating a TRUE/FALSE vector of the test conditions...
  dynamic_check <- c(test_condition1, test_condition2)

  #Creating a vector of the test condition names...
  dynamic_check_names <- c("test_condition1_name", "test_condition2_name")

  #Creating a vector of the test condition names that are TRUE...
  dynamic_check_output <- paste0(dynamic_check_names[dynamic_check], collapse = " ")

  #Outputting to test.results...
  test.results[2, 4] <- paste0("The following condition(s) are true: ",
↪  dynamic_check_output, ". (Any additional feedback as needed.)")
}
```

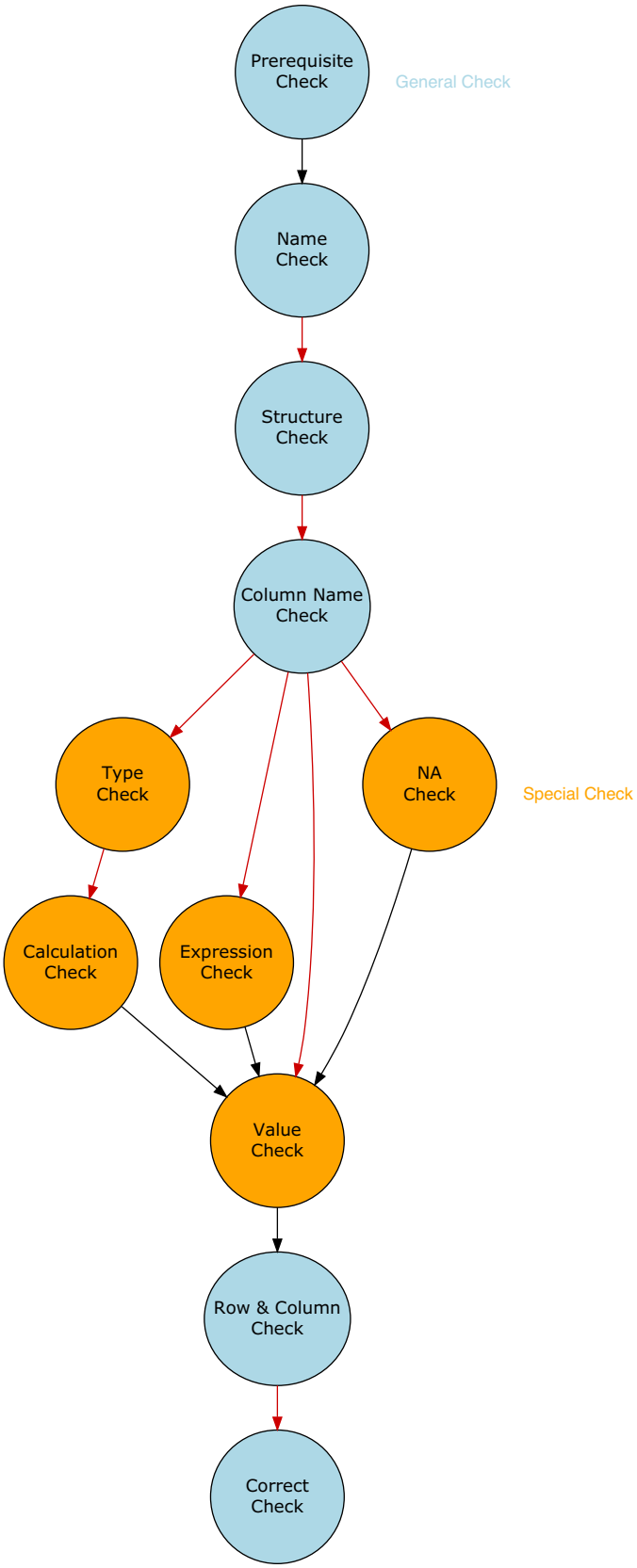
4 Check Flowchart

The flowchart below outlines the recommended order for General and Special Checks when building the autograder script for a Public Question.

Red arrows flowing to a Check indicate its **dependency on previous Checks.**

Note:

This sequence is flexible. Experienced TAs can reorder Checks to suit the logic of a specific question – *so long as dependencies are preserved*.


Prerequisite Check
General Check
Name Check
Structure Check
Column Name Check
Type Check
NA Check
Special Check
Calculation Check
Expression Check
Value Check
Row & Column Check
Correct Check

5 Putting It All Together

The following are examples of autograder code for a Public Question involving a tibble, vector, and list respectively, incorporating many of the General and Special Checks discussed earlier.

5.1 Example: Tibble

Consider the following question from Homework 9.[1]

Part 1: Coding Assignment

4. (*Public Question*) Next, using `un_data_tfr` create a new tibble called **`un_data_tfr_gdp`** that contains a column for the year, a column for the region, and a column for the average GDP per capita in each region for each year. The new column should be called **`gdp_per_capita`**, and should be created by dividing the sum of GDP by the sum of population for each region and year.

The answer should look something like:

```
un_data_tfr_gdp <- un_data_tfr |>
  summarise(gdp_per_capita = weighted.mean(gdp_per_capita, population, na.rm = T),
            .by = c(year, region))
```

Assume this question is worth 20 points and that `un_data_tfr` was created in Question 2.

Then, the autograder code for this question could look like:

```
#Autograder Code for Part 1 Question 4----------

#Initializing `test.results[4, ]`
test.results[4, ] <- c("Part 1 Question 4 (Public)", 0, 20, "Try again. Hint: Start with
↪  summarize().")

#Prerequisite Check
if(test.results[2, 2] == 0){
```

[1] Technically, this is the *fifth* question of the assignment, but I have modified it to be the fourth to align with its question number in `test.results[#, ]`.

```
  test.results[4, 4] <- "This question depends on `un_data_tfr` from Question 2 being
↪  correct. Try again."
#Name Check
} else if(is.error(un_data_tfr_gdp)){
  test.results[4, 4] <- "`un_data_tfr_gdp` is not found. Please make sure to name the
↪  variable correctly. Hint: Check spelling and capitalization."
#Structure Check (for a tibble)
} else if(!is_tibble(un_data_tfr_gdp)){
  test.results[4, 4] <- "`un_data_tfr_gdp` is not a tibble. Please make sure it is a
↪  tibble."
#Column Name Check
} else if(!all(colnames(un_data_tfr_gdp_test) %in% colnames(un_data_tfr_gdp))){
  test.results[4, 4] <- paste0(c("The following column(s) should be in un_data_tfr_gdp,
↪  but they were not found in your answer:",
                         colnames(un_data_tfr_gdp_test)[!(colnames(un_data_tfr_gdp_test)
                           ↪  %in% colnames(un_data_tfr_gdp))],
                                ". Hint: Use summarize() with appropriate .by = argument
                                 ↪  (besides gdp_per_capita, the other two columns that
                                 ↪  you need will be
                                your .by argument, in the form of a vector.)"), collapse
↪  = " ")
} else if(!all(colnames(un_data_tfr_gdp) %in% colnames(un_data_tfr_gdp_test))){
  test.results[4, 4] <- paste0(c("The following column(s) should not be in
↪  un_data_tfr_gdp, but they were found in your answer :",
                                colnames(un_data_tfr_gdp)[!(colnames(un_data_tfr_gdp)
                                 ↪  %in% colnames(un_data_tfr_gdp_test))],
                                ". Hint: Use summarize() with appropriate .by= argument
                                 ↪  (besides gdp_per_capita, the other two columns that
                                 ↪  you need will be
                                your .by argument, in the form of a vector.)"), collapse
↪  = " ")
#NA Check
} else if(sum(is.na(un_data_tfr_gdp$gdp_per_capita)) > 0){
  test.results[4, 4] <- "There are `NA` values in your `gdp_per_capita` column. Hint:
↪  You can do one of the following: (1) filter out all rows with missing gdp (or
↪  gdp_per_capita) and population values first, (2) set na.rm correctly when necessary."
#Type Check (numeric)
} else if(!is.numeric(un_data_tfr_gdp$gdp_per_capita)){
  test.results[4, 4] <- "`gdp_per_capita` should be a numeric column."
#Calculation Check
} else if(isTRUE(all.equal((un_data_tfr_gdp$gdp_per_capita / 1000) |> sort(),
                           un_data_tfr_gdp_test$gdp_per_capita |> sort()))){
```

```
  test.results[4, 4] <- "The `gdp_per_capita` column is incorrect. Hint: If you used
↪  `gdp` to calculate `gdp_per_capita` column in summarize(), don't forget that
↪  `population` is in terms of thousands of people."
#Value Check
} else if(!isTRUE(all.equal(un_data_tfr_gdp$gdp_per_capita |> sort(),
                            un_data_tfr_gdp_test$gdp_per_capita |> sort(),
                            tolerance = 0.001))){
  test.results[4, 4] <- "The `gdp_per_capita` column is incorrect. Hint: Within
↪  summarize(), try weighted.mean(). Or you can use sum() to implement a weighted
↪  average. Note: If you used sum(), you will need to remove rows that have a missing
↪  gdp or population value in the first place."
#Row Check
} else if(nrow(un_data_tfr_gdp) != nrow(un_data_tfr_gdp_test)){
  test.results[4, 4] <- "The number of rows in un_data_tfr_gdp is not correct. Hint: Did
↪  you use the summarize() function correctly?"
#Correct Check
} else if(isTRUE(all.equal(un_data_tfr_gdp |> ungroup() |>
                           select(colnames(un_data_tfr_gdp_test)) |>
                           arrange(across(everything())),
                           un_data_tfr_gdp_test |> ungroup() |>
                           select(colnames(un_data_tfr_gdp_test)) |>
                           arrange(across(everything())),
                           tolerance = 0.001,
                           check.attributes = F))){
  test.results[4, 2] <- 20
  test.results[4, 4] <- "Well done!"
}
```

5.2 Example: Vector

Consider the following question from Homework 2.

Part 1: Coding Assignment

2. (*Public Question*) Using the function `sample`, randomly sample with replacement numbers from **`key`** and save this vector as **`index`**. The length of **`index`** should be 3.

The answer should look something like:

```
index <- sample(key, size = 3, replace = TRUE)
```

Assume this question is worth 10 points and that `key` was created in Question 1.

Then, the autograder code for this question could look like:

```
#Autograder Code for Part 1 Question 2----------

#Initializing `test.results[2, ]`
test.results[2, ] <- c("Part 1 Question 2 (Public)", 0, 10, "Try again.")

#Prerequisite Check
if(test.results[1, 2] == 0){
  test.results[2, 4] <- "This problem depends on Part 1 Question 1 being correct. Try
↪  again."
#Name Check
} else if(is.error(index)){
  test.results[2, 4] <- "`index` is not found. Please make sure your variable is named
↪  correctly."
#Structure Check (for a vector)
} else if(!is.vector(index, mode = "any")){
  test.results[2, 4] <- "Please make sure `index` is a vector."
#Structure Check (for a numeric vector)
} else if(!is.vector(index, mode = "numeric")){
  test.results[2, 4] <- "Please make sure `index` is a numeric vector."
#Length Check (i.e., Row & Column Check for a vector)
} else if(length(index) != length(index_test)){
  test.results[2, 4] <- "Please make sure the vector `index` is the correct length, as
↪  specified by the prompt (i.e., there are 3 elements in the vector)."
#Value Check (for a vector)
} else if(!all(index %in% index_test)){
  test.results[2, 4] <- "`index` has incorrect values. Hint: did you use the argument
↪  replace = TRUE in the `sample()` function?"
#Correct Check
} else if(isTRUE(all.equal(index, index_test,
                           tolerance = 0.001, check.attributes = F))){
  test.results[2, 4] <- "Well done!"
  test.results[2, 2] <- 10
}
```

5.3 Example: List

Consider the following question from Homework 7.

Part 1: Coding Assignment

3. (*Public Question*) Create a table called **`crime_race_counts`** that shows the number of crimes along with the percentage of crimes that were committed against victims of each race in `victim_race`. Your percentages should be out of 100 and rounded to one decimal place. You should remove any rows that do not record the victim's race.

The answer should look something like:

```
crime_race_counts <- crimeData_clean |>
  filter(!is.na(vict_race)) |>
  tabyl(vict_race) |>
  adorn_pct_formatting()
```

Assume this question is worth 10 points and that `crimeData_clean` was created in Question 1.

Then, the autograder code for this question could look like:

```
#Complete Autograder Code for Part 1 Question 3----------

#Initializing `test.results[3, ]`
test.results[3, ] <- c("Part 1 Question 3 (Public)", 0, 10, "Try again. Hint: Look at
↪  the function `janitor::tabyl()`.")

#Prerequisite Check
if(test.results[1, 2] == 0){
  test.results[3, 4] <- "This question depends on Question 1 being correct. Try again."
#Name Check
} else if(is.error(crime_race_counts)){
  test.results[3, 4] <- "`crime_race_counts` is not found. Please make sure the variable
↪  is named correctly."
#Structure Check (for a list)
} else if(!is.list(crime_race_counts)){
  test.results[3, 4] <- "`crime_race_counts` is not a list. Make sure it is a list.
↪  Hint: Look at the function `janitor::tabyl()`."
#Named List Check (i.e., to make sure the list is named)
} else if(is.null(names(crime_race_counts))){
    test.results[3, 4] <- "`crime_race_counts` has no names. Make sure it has names.
↪  Hint: Look at the function `janitor::tabyl()`."
```

```
#NA Check
} else if(NA %in% crime_race_counts$vict_race){
    test.results[3, 4] <- "The column `vict_race` contains `NA` values. Remember to
↪  remove any rows which do not record the victim's race."
#List Name Check (i.e., Column Name Check for a list)
} else if(!all(names(crime_race_counts_test) %in% names(crime_race_counts))){
  test.results[3, 4] <- paste0(paste0(c("The following column name(s) should be in
↪  `crime_race_counts`, but they were not found in your answer:",
                            names(crime_race_counts_test)[!(names(crime_race_counts_test)
                                 ↪  %in% names(crime_race_counts))]),
                                collapse = "  "), ". Hint: Look at the function
                                 ↪  `janitor::tabyl()`.")
} else if(!all(names(crime_race_counts) %in% names(crime_race_counts_test))){
  test.results[3, 4] <- paste0(paste0(c("The following column name(s) should not be in
↪  `crime_race_counts`, but they were found in your answer:",
                                   names(crime_race_counts)[!(names(crime_race_counts)
                                    ↪  %in% names(crime_race_counts_test))]),
                                  collapse = "  "), ". Hint: Look at the function
                                   ↪  `janitor::tabyl()`.")
#Expression Check (for % symbol)
} else if(any(!str_detect(crime_race_counts$percent, "%"))){
  test.results[3, 4] <- "The `percent` column should be formatted as a percentage. Hint:
↪  Look at the function `janitor::adorn_pct_formatting()`."
#Expression Check (for 1 decimal place)
} else if(any(str_detect(crime_race_counts$percent, "\\.\\d{2,}"))){
    test.results[3, 4] <- "The `percent` column should be formatted by 1 decimal place.
↪  Hint: Look at the argument `digits` in `janitor::adorn_pct_formatting()` (note its
↪  default value)."
#Row & Column Check (note: the Column Check is redundant here)
} else if(nrow(crime_race_counts) != nrow(crime_race_counts_test)){
  test.results[3, 4] <- "`crime_race_counts` has the incorrect number of rows. Hint:
↪  Look at the `janitor::tabyl()` function."
} else if(ncol(crime_race_counts) != ncol(crime_race_counts_test)){
  test.results[3, 4] <- "`crime_race_counts` has the incorrect number of columns. Hint:
↪  Look at the `janitor::tabyl()` function."
#Correct Check
} else if(isTRUE(all.equal(crime_race_counts |> ungroup() |>
                           select(names(crime_race_counts_test)) |>
                           arrange(across(everything())),
                           crime_race_counts_test |> ungroup() |>
                           select(names(crime_race_counts_test)) |>
                           arrange(across(everything())),
                           tolerance = 0.001,
```

```
                          check.attributes = F))){
    test.results[3, 2] <- 10
    test.results[3, 4] <- "Well done!"
}
```

6 Private Questions

Private Question Checks are designed to give students *necessary and basic feedback* – not detailed hints. Their purpose is to flag basic issues without guiding the student toward the solution. Accordingly, Private Question Checks only incorporate General Checks.

These Checks help students identify *trivial errors* (e.g., a misspelled column name) and *reduce time debugging*, allowing them to focus on the question's core aspects.

To achieve this, Private Questions are graded using three built-in functions: `private_grader()`, `ggplot_grader()`, and `private_tabyl_grader()`.[1] The following sections explain how to use each function and its parameters.

Note:

Function sample code omit the `test.results[#, ]` initialization for readability.

Remember to *always* initialize `test.results[#, ]` before implementing a question's autograder code.

6.1 private_grader()

Description

Grades Private Questions involving tibbles, vectors, lists, or plots, providing only basic feedback.

The function implements many of the General Checks discussed earlier.

Usage

```
private_grader(var_name,
               var,
```

[1] For ECON 145 TAs, `private_grader()`, `ggplot_grader()`, and `private_tabyl_grader()` can be found in the subfolder `helper_functions/misc_helper_functions`.

```
              var_test,
              status = paste0("Question ", quest_num, " (Private)"),
              var_status = "tib",
              prev_check = TRUE,
              is_tibble_check = TRUE,
              quest_check_read = FALSE,
              quest_num = 0,
              quest_pt = 0,
              quest_prev = 1,
              quest_prev_status = NULL)
```

Arguments

Element	Description	Default
var_name	character, the expected name of var	-
var	expected R object, the student's object	-
var_test	other R object, the answer key's object to be compared with var	-
status	character, the question's displayed part, number, and type (Private/Public); Example: "Part 2 Question 2 (Private)"	paste0("Question ", quest_num, " (Private)")
var_status	the data type of var; Example: "tib" (tibble), "vect" (vector), "lst" (list), "plt" (plot)	"tib"
prev_check	logical indicating if the Prerequisite Check should be triggered. Requires quest_prev value and can optionally be combined with quest_prev_status	TRUE
is_tibble_check	logical indicating if the tibble Structure Check should be triggered	TRUE

quest_check_read	logical indicating if the message "Please make sure you are using read_csv(), not read.csv()." should be attached to the tibble Structure Check	FALSE
quest_num	numeric, the question's number	0
quest_pt	numeric, the question's maximum score	0
quest_prev	numeric, the prerequisite question's number	1
quest_prev_status	character, the prerequisite question's part and number; Example: "Part 2 Question 3b"	NULL

Value

The `test.results[#, ]` vector, containing the question's `status`, amount of points awarded, total point value, and feedback message.

See What is test.results? for more details.

Details

Designed for grading Private Questions involving **tibbles, vectors, lists, or plots only**.

Plot grading via `private_grader()` is limited. Consistent with previous ECON 145 autograders, full credit is awarded if the student simply names the plot properly and gets any prerequisite question correct. For more detailed feedback, consider using the `ggplot_grader()` for ggplot-type questions. See Plots for more information.

The function grades **two-way tabyl lists only**. For three-way tabyl lists, please use `private_tabyl_grader()` instead.

Note that `var` is the *expected* student's R object and may not necessarily exist (e.g., the student saved their variable under a different name).

- This is not a problem because `private_grader()` first performs the Name Check. If `var` does not exist, the subsequent Checks will not be triggered.

Also note that `quest_prev_status` is an *optional* argument when setting `prev_check` and `quest_prev` but helps *clarify* an unclear prerequisite question's location in the feedback message.

- For example, if the prerequisite question is Part 2 Question 3b, but is the eighth question of the assignment (`quest_prev = 8`), this argument can specify its particular location through setting `quest_prev_status = "Part 2 Question 3b"`. Otherwise, its question's number will be displayed (i.e., `"Question 8"`).

Example 1: Tibble

Consider the following questions from Homework 14.

Part 1: Coding Assignment

1. First, you will combine player salary data with CPI data.

   (a) (*Private Question*) Load in the `Master.csv` file, keeping the columns for player ID, birth year, birth country, first name, last name, weight, height, bats, throws, debut date, and final game date. Save the tibble as **`player_master_data`**.

   (b) (*Private Question*) Load in the `Salaries.csv` file into R. Join the `player_master_data` tibble with the data from `Salaries.csv` by `playerID`, keeping only rows with data from both datasets. Save the resulting, joined tibble as **`player_salary_data1`**.

Assume each question is worth 5 points.

Then, the autograder code for this question should look like:[2]

```
#Comparing the student's tibble `player_master_data` with the answer key's
↪  `player_master_data_test`
#Since this is the first question of the assignment, there is no prerequisite check and
↪  so `prev_check` is set to `FALSE`
#Moreover, since this question involves the loading in of a .csv file, it would be
↪  helpful to remind students to use `read_csv()` and not `read.csv()` by setting
↪  `quest_check_read` to `TRUE`
test.results[1, ] <- private_grader("player_master_data", player_master_data,
↪  player_master_data_test,
                                    status = "Part 1 Question 1a (Private)", var_status =
                                      ↪  "tib",
```

[2]While manipulation of the default parameters can allow `private_grader()` to be implemented with fewer explicit inputs, it is recommended to explicitly define *at least* the parameters included in these examples for clarity and consistency.

```
                                    quest_check_read = TRUE, prev_check = FALSE, quest_num
                                      ↪  = 1, quest_pt = 5)

#Comparing the student's tibble `player_salary_data1` with the answer key's
 ↪  `player_salary_data1_test`
#This question relies on the student correctly reproducing `player_master_data` in the
 ↪  previous question, so `prev_check`, `quest_prev`, and `quest_prev_status` are set
 ↪  appropriately
#Since this question also involves the reading in of data, `quest_check_read` is set to
 ↪  TRUE
test.results[2, ] <- private_grader("player_salary_data1", player_salary_data1,
 ↪  player_salary_data1_test,
                                     status = "Part 1 Question 1b (Private)", var_status =
                                       ↪  "tib",
                                    prev_check = TRUE, quest_prev = 1, quest_prev_status =
                                       ↪  "Part 1 Question 1a",
                                     quest_check_read = TRUE, quest_num = 2, quest_pt = 5)
```

Example 2: Vectors

Consider the following questions from Starter Assignment 1.

Part 1: Numeric Object Type and Logical Operators

1. (*Private Question*) Assign number `1.45` to the variable **`num_1`**. Remember that the assignment operator in R is `<-`.
2. (*Private Question*) Assign number `5` to the variable **`num_2`**.

Assume each question is worth 1 point.

Then, the autograder code for this question should look like:

```
#Comparing the student's vector `num_1` with the answer key's `num_1_test`
#Since this question does not rely on another question, `prev_check` is set to `FALSE`
test.results[1, ] <- private_grader("num_1", num_1, num_1_test,
                                     status = "Part 1 Question 1 (Private)", var_status =
                                       ↪  "vect",
                                     prev_check = FALSE, quest_num = 1, quest_pt = 1)

#Comparing the student's vector `num_2` with the answer key's `num_2_test`
#`prev_check` is set similarly
```

```
test.results[2, ] <- private_grader("num_2", num_2, num_2_test,
                                    status = "Part 1 Question 2 (Private)", var_status =
                                     ↪  "vect",
                                    prev_check = FALSE, quest_num = 2, quest_pt = 1)
```

Example 3: Lists

Consider the following questions from Homework 7.

Part 1: Coding Assignment

8. (*Private Question*) Using `crimeData`, create a table called **`age_bin_by_crime_percentages`** that shows what percent of crime experienced by each age bin that is violent or nonviolent. Your table should have a column each for violent and nonviolent crimes and a row for each age bin as well as a row for the total across all age bins.
9. (*Private Question*) Using `crimeData`, create a table called **`crime_by_age_bin_percentages`** that shows the percent of violent and nonviolent crimes that occurred in each age bin. Your table should have a column for each age bin and a row each for violent and nonviolent crimes. You should also have a row for the total for each age bin.

Assume each question is worth 10 points and that `crimeData` was created in Question 6.

Then, the autograder code for these questions should look:

```
#Note: Tables (especially those created with `janitor::tabyl()`) should be classified as
 ↪  lists ("lst") when using `private_grader()`

#Comparing the student's list `age_bin_by_crime_percentages` with the answer key's
 ↪  `age_bin_by_crime_percentages_test`
#Since this question relies on `crimeData` being correct, `prev_check` and `quest_prev`
 ↪  are set appropriately (`quest_prev_status` is omitted)
test.results[8, ] <- private_grader("age_bin_by_crime_percentages",
 ↪  age_bin_by_crime_percentages, age_bin_by_crime_percentages_test,
                                    status = "Part 1 Question 8 (Private)", var_status =
                                     ↪  "lst",
                                    prev_check = TRUE, quest_num = 8, quest_pt = 10,
                                     ↪  quest_prev = 6)

#Comparing the student's list `crime_by_age_bin_percentages` with the answer key's
 ↪  `crime_by_age_bin_percentages_test`
#`prev_check` and `quest_prev` are set appropriately (`quest_prev_status` is omitted)
```

```
test.results[9, ] <- private_grader("crime_by_age_bin_percentages",
↪  crime_by_age_bin_percentages, crime_by_age_bin_percentages_test,
                                    status = "Part 1 Question 9 (Private)", var_status =
                                     ↪  "lst",
                                    prev_check = TRUE, quest_num = 9, quest_pt = 10,
                                     ↪  quest_prev = 6)
```

Example 4: Plots

Background

Previously in ECON 145, plots have not been *rigorously graded* by the autograder due to the desire to award credit to a wide range of students' plots that closely resemble, but are not identical to, the solution.

- Accordingly, `private_grader()` only performs the Prerequisite Check and Name Check when grading plots. If the student's plot bypasses these Checks, full credit is awarded. Afterwards, plots are *manually graded* by the TAs in the students' write-ups.
- `ggplot_grader()` was made in an attempt to provide some level of Checks and feedback for plots created through the `ggplot2` package. See `ggplot_grader()` for details.

Now consider the hypothetical homework question:

Part 1: Coding Assignment

2. (*Private Question*) Using `basketball_data` from Question 1, create a scatter plot of points (`pt`) versus assists (`ast`). Save the plot as **`basketball_scatter`**.

Assume this question is worth 5 points.

Then, the autograder code for this question should look like:

```
#Comparing the student's plot `basketball_scatter` with the answer key's
↪  `basketball_scatter_test`
#Since this question relies on `basketball_data` being correct, `prev_check` and
↪  `quest_prev` are set appropriately (`quest_prev_status` is omitted)
test.results[2, ] <- private_grader("basketball_scatter", basketball_scatter,
↪  basketball_scatter_test,
                                    status = "Part 1 Question 2 (Private)", var_status =
                                     ↪  "plt",
                                    prev_check = TRUE, quest_num = 2, quest_pt = 5,
                                     ↪  quest_prev = 1)
```

Acknowledgements

Author: Riley Berman

Contributors: Alex Zhao, Jack Keefer, Shreya Sinha, Michal Snopek

6.2 ggplot_grader()

Description

Grades Private (or Public) Questions involving `ggplot` objects from the `ggplot2` package [Wickham et al., 2024].

This function provides more detailed feedback than traditionally given by the autograder. Rather than evaluating for full plot correctness, `ggplot_grader()` checks for key plot attributes, like a proper title or x-y aesthetics, and returns targeted feedback. Accordingly, some flexibility is maintained between the student's plot and the solution.

See Plots for more information.

Usage

```
ggplot_grader(var_name,
              var,
              var_test,
              status = paste0("Question ", quest_num, " (Private)"),
              prev_check = FALSE,
              title_check = FALSE,
              axis_check = FALSE,
              aesthetic_check = FALSE,
              aesthetic_flexible_check = FALSE,
              axis_name_check = FALSE,
              data_check = FALSE,
              facet_wrap_check = FALSE,
              geom_check = FALSE,
              geom_check_name = c("GeomPoint"),
              color_fill_check = FALSE,
              color_fill_check_name = c("colour"),
              quest_num = 0,
              quest_pt = 0,
              quest_prev = 1,
              quest_prev_status = NULL)
```

Arguments

Element	Description	Default
var_name	character, the expected name of var	-
var	expected ggplot object, the student's ggplot	-
var_test	other ggplot object, the answer key's ggplot to be compared with var	-
status	character, the question's displayed part, number, and type (Private/Public)	paste0("Question ", quest_num, " (Private)")
prev_check	logical indicating if the Prerequisite Check should be triggered; requires quest_prev value and can optionally be combined with quest_prev_status	FALSE
title_check	logical indicating if an existence check for a graph title should be triggered	FALSE
axis_check	logical indicating if an existence check for the x-y axis labels should be triggered	FALSE
aesthetic_check	logical indicating if a strict check for the correct x-y aesthetic mappings should be triggered; alternative to aesthetic_flexible_check; see aesthetic_check or aesthetic_flexible_check?	FALSE
aesthetic_flexible_check	logical indicating if a flexible check for the correct x-y aesthetic mappings should be triggered; alternative to aesthetic_check; see aesthetic_check or aesthetic_flexible_check?	FALSE
axis_name_check	logical indicating if a check for the correct x-y axis labels should be triggered	FALSE

data_check	logical indicating if a check for the correct data associated with the x-y aesthetics should be triggered; requires aesthetic_check	FALSE
facet_wrap_check	logical indicating if a check for the existence of ggplot2::facet_wrap() should be triggered	FALSE
geom_check	logical indicating if an existence check for ggplot layer types (e.g., ggplot2::geom_point()) should be triggered; requires geom_check_name	FALSE
geom_check_name	character vector of the various ggplot layer types to check for in geom_check; Example: c("GeomPoint", "GeomSmooth")	c("GeomPoint")
color_fill_check	logical indicating if an existence check for the "color" and/or "fill" arguments should be triggered; requires color_fill_check_name	FALSE
color_fill_check_name	character vector of the "color" and/or "fill" arguments to check for in color_fill_check; Example: c("colour", "fill")	c("colour")
quest_num	numeric, the question's number	0
quest_pt	numeric, the question's maximum score	0
quest_prev	numeric, the prerequisite question's number	1
quest_prev_status	character, the prerequisite question's part and number	NULL

Value

The `test.results[#, ]` vector, containing the question's `status`, amount of points awarded, total point value, and feedback message.

See What is test.results? for more details.

Details

Designed for grading Private (or Public) Questions involving **ggplot objects only** (i.e., objects with a `ggplot` class attribute).

The function flexibly grades ggplot objects by allowing the user to specify which ggplot attributes to check for.

Note that `var` is the *expected* student's ggplot object and may not necessarily exist or be a ggplot object (e.g., the student decided to use R's base `plot()` instead).

- This is not a problem because `ggplot_grader()` first tests for a `ggplot` class attribute and then performs the Name Check. If `var` does not exist or is not a ggplot object, the subsequent Checks will not be triggered.

Like in `private_grader()`, `quest_prev_status` is an *optional* argument when using `prev_check` and `quest_prev` but helps *clarify* an unclear prerequisite question's location in the feedback message.

When to use aesthetic_check or aesthetic_flexible_check?

`aesthetic_check` tests if the x-y aesthetic mappings of `var` are *exactly the same* as in `var_test`, while `aesthetic_flexible_check` tests if the x-y aesthetic mappings of `var_test` are *contained* in `var`.

Consider the following motivating example:

```
#student's ggplot object
var <- data |> ggplot(aes(date, death * 100))

#answer key's ggplot object
var_test <- data |> mutate(death = death * 100) |>
            ggplot(aes(date, death))
```

Under `aesthetic_check`, the student's `var` would *not be considered correct*, because its x-y aesthetic mappings (`x = date, y = death * 100`) are not the same as in `var_test` (`x = date, y = death`), despite the ggplots being *virtually identical*.

However, under `aesthetic_flexible_check`, `var` is now *considered correct*, since it contains `var_test`'s aesthetics.

`aesthetic_flexible_check` is more flexible than `aesthetic_check`, since it allows for a wider range of acceptable x-y aesthetics. However, because the student's aesthetics can then vary *slightly* from the answer key, the `data_check` **can't be performed without an error occurring.**[a]

Consequently, `data_check` is *automatically* turned off when `aesthetic_flexible_check` is set to `TRUE`.

[a] We haven't found any smooth way to fix this, which might entail comparing only the final data used in the ggplot without reference to the x-y aesthetics. The problem we ran into is that to compare the student's and answer key's ggplot data, while allowing for flexible row and column ordering, we need to have *exactly* the same x-y aesthetics. Perhaps something like `ggplot_build` can resolve this.

Recommendation: If it is reasonable that a student could use *similar*, but *not identical*, x-y aesthetics to the solution (like above), use `aesthetic_flexible_check`, keeping in mind that `data_check` will be turned off. Otherwise, if the aesthetics can reasonably be expected to be the same as the solution and/or the prompt is explicit about which aesthetics to chose, use `aesthetic_check`. Implementing a `data_check` is available here.

- Generally, opting for `aesthetic_flexible_check` is the *safer and better choice* (at least until this issue is resolved).

Warning:

`aesthetic_flexible_check` and `aesthetic_check` are *mutually exclusive checks*.

Do not use both arguments together – that is, don't set both to `TRUE`.

Example 1

Consider the following question from Homework 10.

Part 1: COVID-19 Data

4. (*Public Question*) Next, create a bar plot, using `ggplot2`, that plots the percentage of people who died from COVID-19 based on their vaccination status and age group. The plot should have four bars in two groups. One group of bars should be for those unvaccinated and the second group of bars should be for those vaccinated. In each group, you should have two bars representing the percentage who died of COVID-19, one for those 50 years or older and one for those younger than 50. Save your bar plot as **`covid_plot2`**.

The answer should look something like:

```
covid_plot2 <- covid_data |>
  summarize(death = sum(outcome == "death")/n(),
            survived = sum(outcome == "survived")/n(),
            .by = c(vaccine_status, age_group)) |>
    ggplot(aes(x = vaccine_status, y = death, fill = age_group)) +
    geom_bar(stat = "identity", position = "dodge") +
    ylab("percentage")
```

Assume this question is worth 20 points and that we want to check if the ggplot has:

- a `geom_bar` layer,
- a `fill` argument,
- and x-y axis labels.

Then, the autograder code for this question should look like:

```
#Comparing the student's ggplot `covid_plot2` with the answer key's `covid_plot2_test`
#Note: the student can use `geom_col()` instead of `geom_bar()`, since it has the same
↪  ggplot attribute `GeomBar`
test.results[4, ] <- ggplot_grader("covid_plot2", covid_plot2, covid_plot2_test,
                                   status = "Part 1 Question 4 (Public)", axis_check = TRUE,
                                       geom_check = TRUE, geom_check_name = c("GeomBar"),
                                       color_fill_check = TRUE, color_fill_check_name =
                                       ↪  c("fill"),
                                       quest_num = 4, quest_pt = 20)
```

Example 2

Consider the following question from Homework 12.

Part 1: Coding Assignment

4. In the final coding question, you will recreate Figures 1 and 2, which represent the median home price in the data for each state and the median home price to county per capita income ratio for each state.

    c) (*Private Question*) Using `priceIncomeData`, create a bar plot called **`priceByIncomePlot`** that shows the median home price to county per capita income ratio for each state and the US as a whole. The x-axis should be the state, and the y-axis should

be the median home price to county per capita income ratio. The bars should be colored based on if the state median home price to county per capita income ratio is above or below the US as a whole.

The answer should look something like:

```
priceByIncomePlot <- priceIncomeData |>
  ggplot(aes(x = reorder(state, price_by_income), y = price_by_income)) +
  geom_bar(stat = "identity", aes(fill = above_below_us_price_by_income)) +
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) + #rotate x-axis values for
   ↪  visibility
  labs(title = 'Median Home Price to Average Income Ratio', x = 'State', y = 'Median
   ↪  Price to Average Income Ratio')
```

Assume this question is worth 15 points and is the eighth question of the assignment.

We want to check if the ggplot has:

- a title,
- the correct x-y axis labels,
- a `geom_bar` layer,
- a `fill` argument,
- and a correctly created `priceIncomeData` from a previous question (the sixth question).

Then, the autograder code for this question should look like:

```
#Comparing the student's ggplot `priceByIncomePlot` with the answer key's
 ↪  `priceByIncomePlot_test`
#Note: `axis_name_check` can be used without `axis_check` to check if the x-y axis
 ↪  labels are correct (and therefore exist)
test.results[8, ] <- ggplot_grader("priceByIncomePlot", priceByIncomePlot,
 ↪  priceByIncomePlot_test,
                                   status = "Question 4c (Private)", title_check = TRUE,
                                    axis_check = TRUE, axis_name_check = TRUE,
                                    geom_check = TRUE, geom_check_name = c("GeomBar"),
                                    color_fill_check = TRUE, color_fill_check_name =
                                     ↪  c("fill"),
                                   prev_check = TRUE, quest_prev = 6, quest_prev_status =
                                     ↪  "Question 4a",
                                    quest_num = 8, quest_pt = 15)
```

Example 3

Consider the following question from Homework 13.

Part 1: Coding Assignment

2. In this question, you will work with the `gdpData.csv` file to analyze national Olympic performance and GDP.

    d) (*Public Question*) Next, using `olympic_gdp_data` and `ggplot`, create a group of scatter plots that plot the percentage of the GDP and percentage of the medals won by each country in the 2010 through 2016 Olympic games. The x-axis should contain the percentage of the GDP and the y-axis should contain the percentage of the medals won. The plots should be faceted by the year and season of the Olympic games, such that one plot is for the 2010 winter Olympics, another is for the 2012 summer Olympics, etc. Include a line with slope equal to 1 to denote the values for which the percent of GDP and percent of medals won is equal. You should also label the points on your graph with the NOC codes for the countries that have at least 5% of the GDP or medals won during each Olympic games. Save your plot as **`olympic_gdp_perc_plot`**.

The answer should look something like:

```
olympic_gdp_perc_plot <- olympic_gdp_data |>
    filter(year >= 2010) |>
    filter(!is.na(gdp_per_capita), medalsPerc > 0.001) |>
    ggplot(aes(x = gdpPerc, y = medalsPerc)) +
    geom_point() +
    geom_text(aes(label = if_else(medalsPerc > 0.05 | gdpPerc > 0.05, noc, ''),
                  hjust = 0, vjust = 0)) +
    geom_abline(intercept = 0, slope = 1, col = 'red') +
    facet_wrap(year~season)
```

Assume that this question is worth 5 points and is the ninth question of the assignment.

We want to check if the ggplot has:

- x-y axis labels,
- the flexibly correct x-y aesthetics,
- `geom_point`, `geom_abline`, and `geom_text` layers,
- a `facet_wrap()` function,
- and a correctly created `olympic_gdp_data` from a previous question (the eighth question).

Then, the autograder code for this question should look like:

```
#Comparing the student's ggplot `olympic_gdp_perc_plot` with the answer key's
↪  `olympic_gdp_perc_plot_test`
#Note: `geom_abline()` can be checked through the `GeomAbline` attribute and `geom_text`
↪  through the `GeomText` attribute
test.results[9, ] <- ggplot_grader("olympic_gdp_perc_plot", olympic_gdp_perc_plot ,
                                   olympic_gdp_perc_plot_test, status = "Question 2d
↪  (Public)",
                                   axis_check = TRUE, aesthetic_flexible_check = TRUE,
                                   geom_check = TRUE, geom_check_name = c("GeomPoint",
                                    ↪  "GeomAbline", "GeomText"),
                                   quest_num = 9, quest_pt = 5, facet_wrap_check = TRUE,
                                  prev_check = TRUE, quest_prev = 8, quest_prev_status =
                                   ↪  "Question 2c (Public)")
```

Example 4

Consider the hypothetical homework question:

Part 1: Coding Assignment

2. (*Private Question*) Using `basketball_data` from Question 1, create a scatter plot of points versus assists with `ggplot`. The x-axis should be `pt` and the y-axis should be `ast`. Save the ggplot as **`basketball_scatter`**.

The answer should look something like:

```
basketball_scatter <- basketball_data |>
  ggplot(aes(x = pt, y = ast)) +
  geom_point()
```

Assume this question is worth 5 points and that we want to check if the ggplot has:

- the correct x-y axis labels,
- the correct x-y aesthetics,
- the correct x-y data,
- a `geom_point` layer,
- and a correctly created `basketball_data` from a previous question (the first question).

Then, the autograder code for this question should look like:

```
#Comparing the student's plot `basketball_scatter` with the answer key's
↪  `basketball_scatter_test`
test.results[2, ] <- ggplot_grader("basketball_scatter", basketball_scatter,
↪  basketball_scatter_test,
                                   status = "Part 1 Question 2 (Private)",
                                    ↪  axis_name_check = TRUE,
                                   aesthetic_check = TRUE, data_check = TRUE,
                                  geom_check = TRUE, geom_check_name = c("GeomPoint"),
                                   quest_num = 2, quest_pt = 5, prev_check = TRUE,
                                    ↪  quest_prev = 1)
```

Acknowledgements

Author: Riley Berman

Contributors: Alex Zhao, Jack Keefer, Shreya Sinha, Michal Snopek

6.3 private_tabyl_grader()

Description

Grades Private Questions involving three-way `tabyl` objects from the `janitor` package [Firke, 2024a].

This function implements many of the General Checks discussed earlier.

Usage

```
private_tabyl_grader(var_name,
                     var,
                     var_test,
                     status = paste0("Question ", quest_num, " (Private)"),
                     prev_check = TRUE,
                     quest_num = 0,
                     quest_pt = 0,
                     quest_prev = 1,
                     quest_prev_status = NULL)
```

Arguments

Element	Description	Default
var_name	character, the expected name of var	-
var	expected tabyl object, the student's tabyl	-
var_test	other tabyl object, the answer key's tabyl to be compared with var	-
status	character, the question's displayed part, number, and type (Private/Public)	paste0("Question ", quest_num, " (Private)")
prev_check	logical indicating if the Prerequisite Check should be triggered; requires quest_prev value and can optionally be combined with quest_prev_status	TRUE
quest_num	numeric, the question's number	0
quest_pt	numeric, the question's maximum score	0
quest_prev	numeric, the prerequisite question's number	1
quest_prev_status	character, the prerequisite question's part and number	NULL

Value

The `test.results[#, ]` vector, containing the question's `status`, amount of points awarded, total point value, and feedback message.

See What is test.results? for more details.

Details

Designed for grading **three-way tabyl objects only** (i.e., objects with a `tabyl` class attribute).

Use `private_grader()` for two-way tabyl objects.

As with the other functions, `var` may not necessarily exist or be a tabyl object.

- This is not a problem because `private_tabyl_grader()` first tests for a `tabyl` class attribute and then performs the Name Check.

Likewise, `quest_prev_status` is an optional argument when using `prev_check` and `quest_prev` but helps *clarify* an unclear prerequisite question's location in the feedback message.

What is a three-way tabyl?

A three-way tabyl is a multi-table created from three variables that is produced by the `janitor::tabyl()` function.

A three-way tabyl will typically create two "sub-tables."

Consider the following example from the `tabyl` documentation [Firke, 2024b]:

```
t3 <- starwars |>
  filter(species == "Human") |>
  tabyl(eye_color, skin_color, gender)

#The result is a tabyl of eye color x skin color, split into a list by gender
```

```
## $feminine
##  eye_color dark fair light none pale tan white
##       blue    0    2     1    0    0   0     0
##  blue-gray    0    0     0    0    0   0     0
##      brown    0    1     3    0    0   0     0
##       dark    0    0     0    0    0   0     0
##      hazel    0    0     1    0    0   0     0
##    unknown    0    0     0    1    0   0     0
##     yellow    0    0     0    0    0   0     0
##
## $masculine
##  eye_color dark fair light none pale tan white
##       blue    0    7     2    0    0   0     0
##  blue-gray    0    1     0    0    0   0     0
##      brown    3    4     3    0    0   2     0
##       dark    1    0     0    0    0   0     0
##      hazel    0    1     0    0    0   0     0
##    unknown    0    0     0    0    0   0     0
##     yellow    0    0     0    0    1   0     1
```

Example

Consider the following modified question from the Final.

Part 3: Creating Tables and Figures

1) (*Private Question*) Using `schoolData`, create a table that shows the percentage of schools that provide free lunch to all students, and how for each year in each `locale` category (City, Suburb, Town, and Rural). Make sure you remove any rows with missing `national_school_lunch_program` or `locale` values before creating your table. Save this table as **`nslp_locale_table`**.

The answer should look something like:

```
#Note: This has been slightly modified from the actual solution
nslp_locale_table <- schoolData |>
  filter(!is.na(locale) & !is.na(national_school_lunch_program)) |>
  janitor::tabyl(year, national_school_lunch_program, locale) |>
  janitor::adorn_percentages() |>
  janitor::adorn_pct_formatting()
```

Assume this question is worth 5 points, is the sixteenth question of the assignment, and that `schoolData` was created in Part 1 Question 1b (the second question).

Then, the autograder code for this question should look like:

```
#Comparing the student's three-way tabyl `nslp_locale_table` with the answer key's
 ↪  `nslp_locale_table_test`
test.results[16, ] <- private_tabyl_grader("nslp_locale_table", nslp_locale_table,
 ↪  nslp_locale_table_test,
                                           status = "Part 3 Question 1 (Private)",
                                            ↪  prev_check = TRUE,
                                           quest_prev = 2, quest_prev_status = "Part 1
                                            ↪  Question 1b",
                                           quest_num = 16, quest_pt = 5)
```

Acknowledgements

Co-authors: Michal Snopek, Riley Berman

Contributors: Alex Zhao, Jack Keefer, Shreya Sinha

7 DGP.R

In the autograder system, the `DGP.R` file assigns each student a unique dataset drawn from the `masterdata` according to their PERMID.

See `00-manual1.pdf` and the PERMID Check for more information.

Note:

The Head TA determines the sampling method for each assignment. TAs are *only* responsible for implementing this sampling technique in `DGP.R`.

In some cases, the file does not require any edits.

7.1 Examples

A few examples of `DGP.R` are provided below.

Example 1

Description: Randomly assign each student 1,000 observations sampled from `masterdata`.

```
DGP <- function(masterdata = "Masterdata.csv", PERMID = PERMID, dataname = dataname){
  masterdata <- read_csv(masterdata)
  set.seed(PERMID)
  #---------------#
  #Edit made here...
  d <- masterdata[sample(1:nrow(masterdata), 1000, replace = F),]
  #---------------#
  readr::write_csv(d, dataname)
}
```

Example 2

Description: Assign the full `masterdata` to each student (no sampling). This results in all students receiving the same dataset (adapted from Homework 4).

```
DGP <- function(masterdata = "Masterdata.csv", PERMID = PERMID, dataname = dataname){
  masterdata <- read_csv(masterdata)
  set.seed(PERMID)
  #---------------#
  #Edit made here...
  d <- masterdata
  #---------------#
  readr::write_csv(d, dataname)
}
```

Example 3

Description: Randomly assign each student 20,000 observations sampled from every `masterdata` *except* for the `masterdata` corresponding to `datasetB.csv`.[1]

For the `masterdata` corresponding to `datasetB.csv`, assign each student the full `masterdata`. Assume that `datasetB.csv` has exactly two columns, `col_1` and `col_2` (adapted from Homework 7).

```
DGP <- function(masterdata = "Masterdata.csv", PERMID = PERMID, dataname = dataname){
  masterdata <- read_csv(masterdata)
  set.seed(PERMID)
  #---------------#
  #Edit made here...
  #No sampling for datasetB.csv...
  if(all(colnames(masterdata) %in% c("col_1", "col_2"))){
    d <- masterdata
  #Everything else is randomly sampled...
  } else{
    d <- masterdata[sample(1:nrow(masterdata), 20000, replace = F),]
  }
  #---------------#
  readr::write_csv(d, dataname)
}
```

Example 4

Description: For the following `masterdata`, assign as follows:

[1] Sometimes, multiple csv files are required for an assignment, creating multiple `masterdata`. The `DGP.R` file will then loop through each `masterdata`.

- `data1.csv`: Randomly sample 10,000 rows.
- `data2.csv`: Randomly sample 20,000 rows.
- `data3.csv`: Assign the full dataset, arrange by its column `data3_col2` (so every student will get their dataset in this format).
- `data4.csv`: Assign the full dataset.

Assume `data1.csv` has some unique column `data1_unique_column` and that `data2_col1, ... ,data2_col5` are the *only* columns in `data2.csv` and `data3_col1` and `data3_col2` are the *only* columns in `data3.csv` (adapted from Homework 14).

```
DGP <- function(masterdata = "Masterdata.csv", PERMID = PERMID, dataname = dataname){
  masterdata <- read_csv(masterdata)
  set.seed(PERMID)
  #---------------#
  #Edit made here...
  #`data1.csv` randomly sampled for 10,000 observations
  if("data1_unique_column" %in% colnames(masterdata)){
    d <- masterdata[sample(1:nrow(masterdata), 10000, replace = F),]
  #`data2.csv` randomly sampled for 20,000 observations
  } else if(all(colnames(masterdata) %in% c("data2_col1", "data2_col2",
                                            "data2_col3", "data2_col4", "data2_col5"))){
    d <- masterdata[sample(1:nrow(masterdata), 20000, replace = F),]
  #`data3.csv` is not sampled and is arranged by `data3_col2`
  } else if(all(colnames(masterdata) %in% c("data3_col1", "data3_col2"))){
    d <- masterdata %>% arrange(data3_col2)
  #`data4.csv` is not sampled
  } else{
    d <- masterdata[sample(1:nrow(masterdata), nrow(masterdata), replace = F),]
  }
  #---------------#
  readr::write_csv(d, dataname)
}
```

Example 5

Description: Assign each student a dataset containing observations from 140 randomly selected countries from `masterdata`'s `Country or Area` column (adapted from Homework 8).

```
DGP <- function(masterdata = "Masterdata.csv", PERMID = PERMID, dataname = dataname){
  masterdata <- read_csv(masterdata)
  set.seed(PERMID)
  #---------------#
  #Edit made here...
  #Creating a list of all the countries in `masterdata`
  country_list <- unique(masterdata$`Country or Area`)
  #Randomly selecting 140 countries
  country_sample <- sample(country_list, 140)
  #Selecting the rows from `masterdata` that contain the selected countries
  d <- masterdata[masterdata$`Country or Area` %in% country_sample, ]
  #---------------#
  readr::write_csv(d, dataname)
}
```

Bibliography

Berry Boessenkool. *berryFunctions: Function Collection Related to Plotting and Hydrology*, 2024. URL https://github.com/brry/berryFunctions. R package version 1.22.5.

Sam Firke. *janitor: Simple Tools for Examining and Cleaning Dirty Data*, 2024a. URL https://github.com/sfirke/janitor. R package version 2.2.1.

Sam Firke. *tabyls: a tidy, fully-featured approach to counting things*, 2024b. URL https://cran.r-project.org/web/packages/janitor/vignettes/tabyls.html. Vignette from the janitor package.

Hadley Wickham and Garrett Grolemund. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, 2016. URL https://r4ds.had.co.nz/. Chapter on Tibbles: https://r4ds.had.co.nz/tibbles.html.

Hadley Wickham, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, Dewey Dunnington, and Teun van den Brand. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*, 2024. URL https://ggplot2.tidyverse.org. R package version 3.5.1.