**Documentation Comenius Project / CASWIC**

Jelmer Jansen

April 2021

Translated and updated by Jules L. Ellis with help of ChatGPT 4, December 2023

# Introduction

The purpose of this document is to describe the internal structure of the Comenius coaching application so that future programmers can maintain and possibly expand it. The document begins with an outline of the program's structure, then describes the routing file, followed by the enum classes, the controller class, the assignments, and the scan functions. If you have any questions about this, you can send them to jelmer- ja@hotmail.com.

# Program Structure

The application consists of a routing file (routes.py) and three main classes, all of which will be discussed in more detail. Since Flask lacks functionality to maintain internal states, the state of the application is loaded from a JSON file when the user sends a request, and saved back into the same file after the application has modified this state.

The functionality of the application is based on a rotating structure, where the user first selects the language in which they want to do assignments. Then they are referred to a screen with multiple dropdown bars, where they can choose what type of assignment they want to do and what type of report. There are three types of reports, two modes for basic reports (practice mode and exam mode), and one for concise reports. When the user indicates they are finished with their report, they are brought back to the selection screen.

## 2.1 Practice Mode

In practice mode, the user answers all questions of a report one by one. If they give a wrong answer, the application provides feedback in the form of a list of elements that are still missing from the answer. The feedback appears between the text field and the question. If the user gives a correct answer, they are automatically referred to the next question.

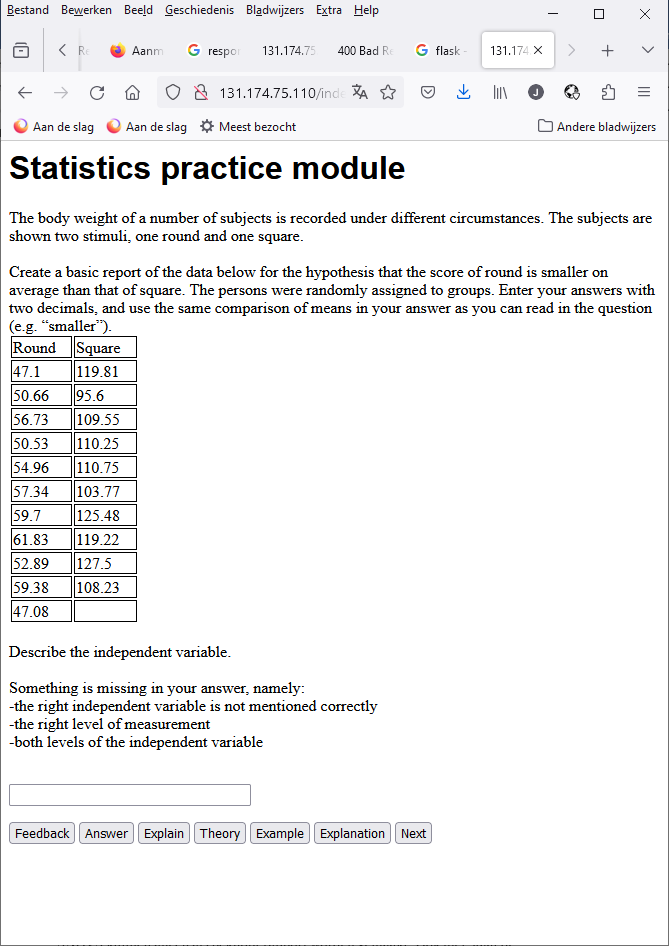


Figure 1: Feedback in practice mode.

The user always has the option to skip a question or to go back to a previous question before the report is completed.

The application uses the file “index.html” to render the webpage in practice mode. The different fields are visible or invisible depending on the input at the parameter “submit field”.

## 2.2 Exam Mode

In exam mode, the user fills in a form and all questions of the report are presented to the user at the same time. They can be filled in separate text fields. Feedback and standard answers are also given for all answers at the same time when the user presses the buttons for this at the bottom of the screen. These are located under the corresponding text field of the question and are colored blue to distinguish them from the black instructions.

## 2.3 Concise Report

In concise reports, the user receives an assignment that is partly filled in, and they are supposed to summarize the conclusion of this assignment in natural language in a text field. About half of the analyses, including MANOVA, ANCOVA, multiple regression, and multivariate repeated-measures ANOVA, can only be made as concise reports. Here too, the feedback is located under the text field and is colored blue.

# Routing

When a user loads a page, either by entering the URL of the application or by pressing a button within the application, the response is defined in the routes.py file. This file specifies how the application should respond to each button on every form. Sometimes the program can directly redirect the user to a new HTML form, but often it must consult the internal state of the application associated with the user.

The file controller.json contains a dictionary of states, all linked to a hash of the respective user's IP address through a dictionary structure. The way these states are adjusted depends on both the existing state and the user's input. When the application is ready to send a response to the user, the state is saved again.

The feedback or default answer, placed under a certain text field when the user requests it, is passed to the forms with the parameters “display” and “displays” from routes.py. This text is sometimes displayed in blue to distinguish it from the instructions associated with the assignment. Other important parameters for these HTML forms are “instruction” (the assignment's instruction in text form, including the data), “varnames” (the names of the variables that are pasted into the tables within the form), and “title” (the title of the application in Dutch or English). The layout of the website is determined by four different HTML files, which are now described.

# HTML Files

The layout of the application is determined by four HTML files. The Python equivalents of these objects are defined as classes in forms.py, which are called in routes.py each time the application sends a page to the user.

## 4.1 index.html

This page is used for the intro screen, the selection menu, and the practice mode of basic reports. Within the HTML file, all components associated with these (the dropdown boxes of the selection screen, the large and small text fields, and the buttons of the practice mode) are written, but what the page displays exactly depends on the "submit\_field" parameter given to the page.

## 4.2 largeform.html and smallform.html

The largeform.html form is used for the exam mode of ANOVA and RMANOVA, and the smallform.html form for the exam mode of *t* tests. Both forms consist of a long list of repetitions of three types of objects: a piece of text for the question's instruction, a text field for the answer, and another piece of text (in blue) for feedback, explanation, or the standard answer. The instruction text is set in routes.py, and the feedback text is first requested from the controller class and then passed to the form.

Both forms contain small variations for different types of assignments in which questions are visible or not. The form decides to display these or not based on the "form\_shape" parameter.

## 4.3 reportform.html

This is the simplest form, and unlike the other forms, the same objects are always visible here (this does not depend on the parameters given). The form contains a piece of text for the assignment, a large text field for the student's answer, and another piece of text below it. Under the text fields are buttons for "Continue," "Feedback," "Answer" (the standard answer), and various types of explanations.

# The Controller Class

The "Controller" class in the interface.py file is used to keep track of the application's state for individual users. Each time a user presses a button, this state is loaded from a JSON dictionary based on the user's IP address. If there is no instance of "Controller" in this dictionary yet, it is created using the "reset()" function, which is described below.

The controller's main attributes are also introduced here. "Assignments" and "sfs" are two separate classes containing functions for generating assignments and evaluating answers, and these will be discussed in more detail later. "Skipable", "prevable", and "answerable" are booleans that indicate whether different buttons should be made visible in the current state. "Formmode" indicates whether the application should show the user a different HTML form than index.html. Self.assignment and self.solution are dictionaries containing the parameters and answers of the current assignment (in a concise report, both are in one dictionary). Self.submit\_field and self.analysis\_type contain enum values, which will be further explained in the next chapter.

Self.protocol indicates which protocol the program is currently working with. A protocol consists of a list of tuples, each tuple containing information for one state. These states apply only to the practice mode of concise reports and menus, and within the assignments in practice mode, a state describes one of the questions. These tuples consist of five values: the text of the question, the example answer, an enum defining the question type, the function for evaluating the answer, and a list of parameters for this function (self.solution is given by default with each function and is thus not in this list). Self.index indicates the current step within the protocol. An example of a protocol can be seen below.

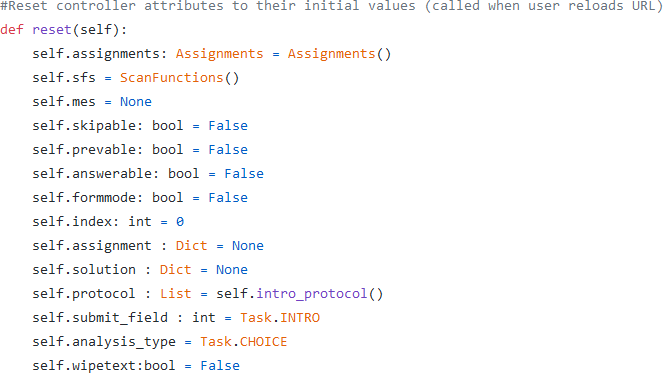


Figure 2: The attributes of the controller class.

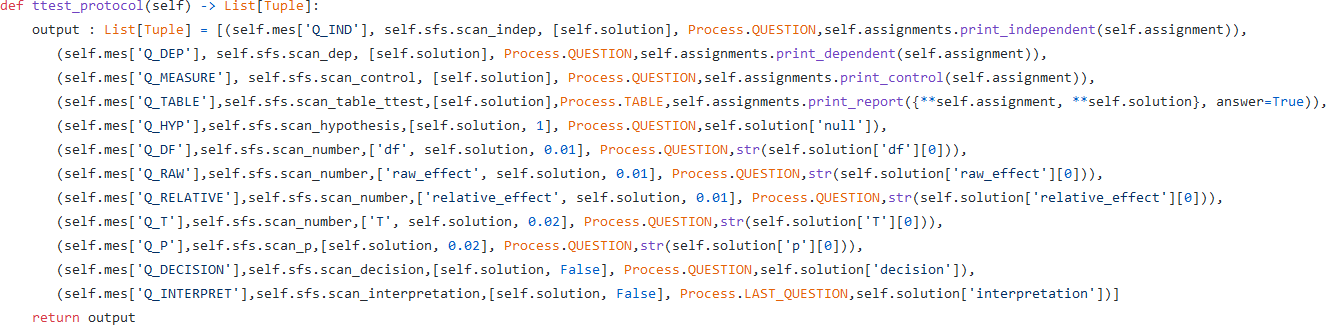


Figure 3: The protocol for the *t* tests.

Self.mes is one of the most important variables within the application. This variable contains a huge dictionary with Dutch or English versions of certain texts, loaded from the file texts.csv when the user chooses the language in which they want to use the application. All keys and almost all values within this dictionary are strings: The only exceptions are "L\_ENGLISH" (a boolean indicating whether the texts are currently in English) and the texts for the various explanation buttons, which are included in self.mes through layered dictionaries.

## 5.1 Enums

The program incorporates two enum classes: Process and Task. These provide a simple way to indicate the current phase of the program. The Process enum consists of six values that describe different phases in using the application. The phases within a specific protocol (like creating an ANOVA report) are stored in the “self.protocol” attribute of the Controller class. The “self.index” attribute determines how far the program has progressed in the protocol. Process values only describe phases for image.html. When the program is using another HTML form, the current Process value refers to the phase the program will be in when using “index.html”. The phases are as follows:

* INTRO - The first page, where the user chooses a language.
* QUESTION - The user answers a question in practice mode.
* LAST\_QUESTION - The last question of an assignment in practice mode.
* CHOOSE\_ANALYSIS - The user can choose from different types of assignments and reports.
* TABLE - A question in an basic report in practice mode where the user must fill in a table.
* FINISH - The user has just completed an assignment and sees a screen with some information; they can return to the selection screen by clicking “Next”.

The Task enum serves a dual function in the program: firstly, it differentiates between various types of assignments by storing the current assignment type in the "assignment type" attribute of each assignment. Secondly, it determines what kind of input field the user should see currently. This dual implementation as a category is because the type of input often depends on the type of analysis. The current Task value is maintained in the "submit field" attribute of the Controller class. The possible Task values are:

* TEXT\_FIELD - A simple one-line text input field.
* TTEST\_BETWEEN - *t* test for independent samples.
* TTEST\_WITHIN - *t* test for paired samples.
* ONEWAY\_ANOVA - 1-way ANOVA.
* TWOWAY\_ANOVA - 2-way ANOVA.
* WITHIN\_ANOVA - Repeated measures ANOVA.
* MREGRESSION - Multiple regression analysis.
* REPORT - Overlapping category for all concise reports.
* INTRO - The "next" button on the introduction screen.
* CHOICE - The dropdown bars on the selection screen.
* TEXT\_FIELD\_LARGE - A large, multi-line text input field for decisions and causal interpretations.
* MANOVA - The MANOVA analysis.
* ANCOVA - The ANCOVA analysis.
* MULTIRM - Multivariate repeated measures ANOVA.
* MULTIRM2 - Double multivariate repeated measures ANOVA (not implemented).
* FINISHED - The "next" button upon completing an assignment.

## 5.2 Functions

The key functions within the controller class are update(), update\_form(), update\_form\_anova(), and update\_form\_report(). They respectively provide feedback on the user input for the four different forms. In practice mode (the update() function), only the current question's evaluation function is called by combining the current protocol with self.index. In exam mode, functions for different questions are called on the respective text fields, and their output is returned as a list of feedback strings for each question.

In concise reports, a higher-order evaluation function is called depending on the analysis chosen by the student, and the output of all the evaluation functions it calls is concatenated into one string, which is then returned and placed under the text field on the webpage.

Moreover, the controller class includes several functions starting with “answer\_”. These work similarly to the feedback functions, but provide the example answer for each question instead. There are also functions starting with “explain”. These determine the output of the other four buttons, based on user input and the attributes of the current instance from self.mes.

# The Assignments

The file assignments.py contains functions for creating, solving, and printing answers. They are in a separate class, Assignments, so all functions can use the same version of self.mes without having to pass this dictionary as a parameter each time.

Besides the Assignments class, there is another class, Variable, for instantiating dependent and independent variables for assignments. This class includes attributes for the variable name and its synonyms, and for independent variables, also the levels and their synonyms. The possible variables are predefined in the file "variables.csv", and all are read and stored in the variable self.variables when a new instance of Assignments is created. The functions get\_factor(), get\_dependent(), and get\_covariates() select a random variable of a certain type when creating an assignment.

Assignment creation occurs in functions starting with “self.create”. Each type of analysis has a separate function, and the assignments of different analyses are distinguished by the integer variable “assignment\_type”.

In the “create\_” functions, only the variables and data points (or their averages) are defined and then returned as a dictionary. The calculation of numerical answers occurs in functions starting with self.solve\_, which take the assignment dictionaries as a starting point and then add all these answers to them.

For concise reports, the process is slightly different: in self.create report, the output of the create and solve functions for that analysis are combined into one dictionary. Thus, concise reports do not have a separate solve function, but they do have a separate print function, as often different tables need to be printed for concise reports.

The functions starting with “self.print\_” return the instructions and data of the corresponding assignment as a long string value, which is later directly pasted onto the HTML page (hence the presence of certain HTML tags like <br>).

# Scan Functions

Similar to assignments.py, scanfunctions.py contains all functions in one class (named ScanFunctions) so they can refer to the same copy of self.mes. A key rule in this class is that functions starting with "scan\_" are higher-order functions that can be called independently to evaluate a text field, while those with "detect\_" are lower-order functions, called only by other evaluation functions. We will now discuss the various evaluation functions. The higher-order scan functions receive the answer text that has to be evaluated and the solution dictionary of the current assignment.



Figure 4: Scan\_indep(), a typical higher order scan function. All these functions begin with ”scan”.

## 7.1 Higher-Order Scan Functions

The first functions in the file can be directly called to evaluate user responses for a specific text field and follow a standard structure. The output is always in the form of [string, bool], where the string contains a list of feedback points about missing elements in the user's answer. If the user has included all correct points, the string contains a brief message indicating the assignment is correctly completed. The bool value returned is False if the user's answer contains all required elements. During practice mode, this bool value is used to decide if the program can proceed to the next part of the protocol. Most higher-order scan functions define a dictionary variable named "scorepoints" at the top. Each pair in this dictionary represents the presence of an element in the answer, such as the independent variable or the "H0:" sign, or even more abstract structures like the order of characters in a hypothesis. Each key in scorepoints gets a True or False value, depending on whether the corresponding element is in the student's answer.

At the bottom of the function is a list of if-statements that check each pair in scorepoints to see if the user has included the necessary element in their answer, adding an associated string from self.mes if not. In some cases, an element in scorepoints may contain a list of bools instead of a single bool, like when checking which population means are included in an answer with a hypothesis. In such cases, the corresponding if-statement is phrased differently.

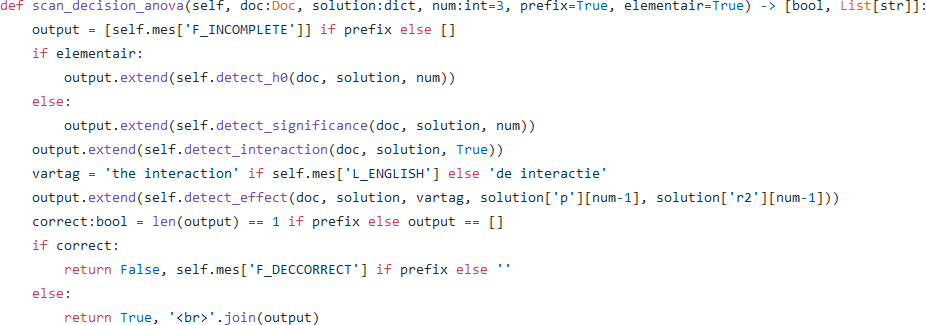


Figure 5: Scan\_decision\_anova(), a function for evaluating the interaction decision. In this example, several lower-order scan functions are invoked, such as detect\_h0() and detect\_interaction().

## 7.2 Higher-Order Scan Functions with SpaCy

The scan functions for decisions and causal interpretations largely follow the same structure as the others, but with a key difference: they receive the student's answer as a "Doc" type instead of a string. The "Doc" type, defined in the SpaCy Python package, is an annotated form of text containing useful information like part-of-speech and dependency tags (currently, the program only uses dependency tags).

The user's input is annotated with a language model loaded from SpaCy. For basic reports, this occurs at the start of the update() functions in interface.py, and for concise reports, in the corresponding split\_grade functions described below. Decisions and causal interpretations, having a more complex structure, involve calling many more lower-order scan functions than those above line 484.

## 7.3 Scan Functions for Concise Reports

The functions used to evaluate concise reports all start with split\_grade, and they return a string value containing all feedback points the user needs to pay attention to. This is achieved by sequentially calling all higher-order functions related to the elements that should be included in the report. For instance, in a *t* test, this includes the name of the analysis, the design, the *t* value, the *p* value, and the decision.

Often, not the entire text of the answer can be provided to each scan function to avoid confusion. For example, a function evaluating the decision for a specific variable might not function correctly if it receives all decisions from the report as input, potentially due to confusion about the number of negations in the input.

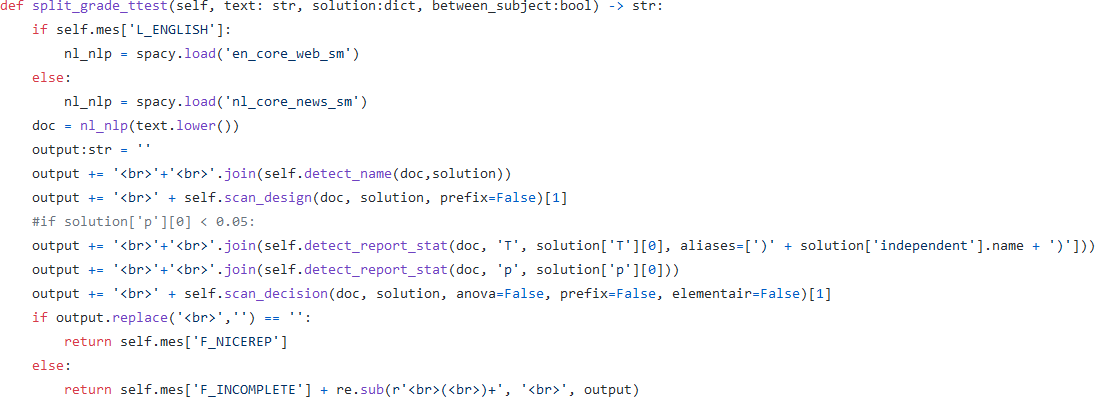


Figure 6: Split\_grade\_ttest(), one of the most simple functions for concise reports.

For all critical higher-order functions, the appropriate sentence is thus identified within the evaluation functions for concise reports to pass to the lower-order functions. This is done by first splitting the total input into sentences, and then passing the sentence that meets the presence or absence of certain keywords. If such a sentence is not in the answer, a message is added to the output indicating that this part of the report is missing.

## 7.4 Lower-Order Scan Functions

Lower-order scan functions differ structurally from higher-order ones. They start with detect\_ instead of scan\_, and instead of returning a boolean and a string, they return a list of strings. Each string represents a pair in scorepoints, indicating elements that are missing in the student's answer. These strings are then later combined in the higher-order scan functions with the output from other lower-order functions to form an organized list.

The internal structure of these functions is similar to higher-order ones, often featuring a scorepoints dictionary tracking the required elements of the answer. A notable difference is that lower-order functions are typically lengthy and contain numerous lists of keywords, as they need to recognize many different ways a student might formulate a part of their answer.

## 7.5 Auxiliary Functions

This section includes functions that are essential for evaluating answers but don't provide direct feedback points. The function check\_causality() examines the influence of an independent variable on a dependent variable using their dependency tags. Distance() calculates the distance between the location of a specific word in two different sentences. Negation\_counter() counts the number of negations (e.g., "not", "no") in the given sentence.

Descendants() is a recursive function that identifies all words under a given word in the sentence's dependency structure. Lef() takes two lists of words and returns True if the Levenshtein distance between any two-word combination from these lists is less than 2, used to recognize variables despite spelling errors. Additionally, there are two functions, sim() and sim\_p(), for evaluating tables filled by the user. Sim\_p() considers the user's answer correct if they provide a correct boundary value for *p* instead of the exact value (e.g., ≤ 0.05 instead of 0.03).

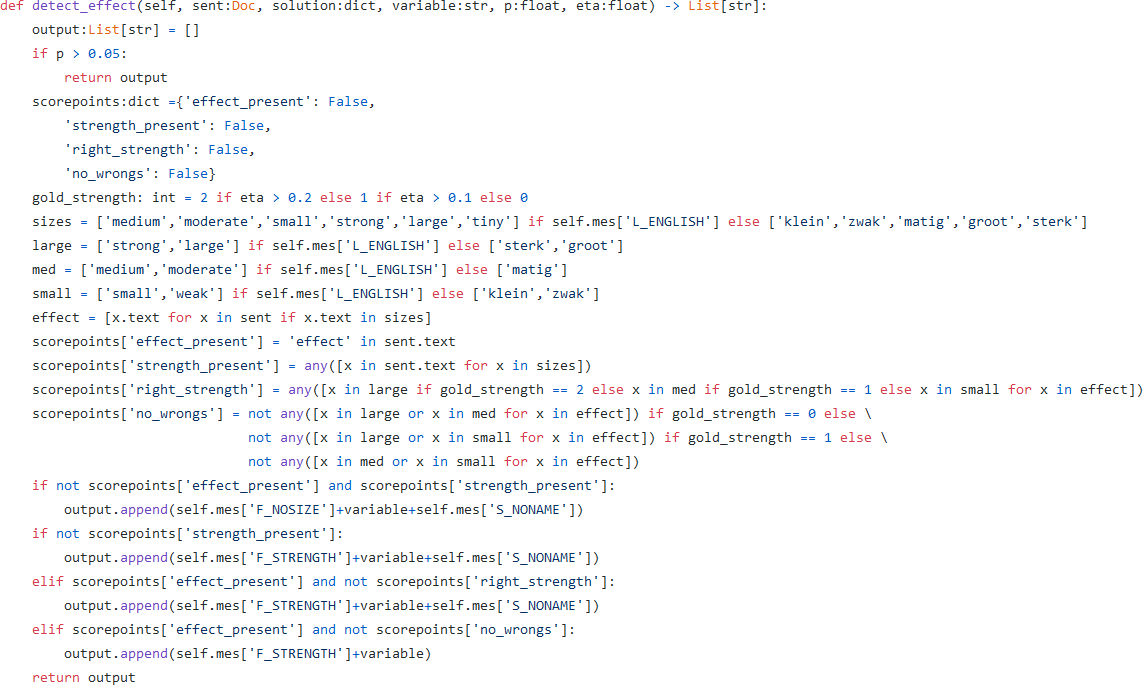


Figure 7: Detect\_effect(), a typical lower order scan function.

## 7.6 Testing Functions

At the bottom of the file are two functions not used by the program itself, but available for the programmer to experiment with SpaCy dependency tags. Print\_dissection() displays the dependency tags of each word in a sentence provided by the user. Load\_dissection() shows the hierarchical structure of the input sentence. These functions are designed to facilitate easier experimentation with the nuances of SpaCy's language processing capabilities.

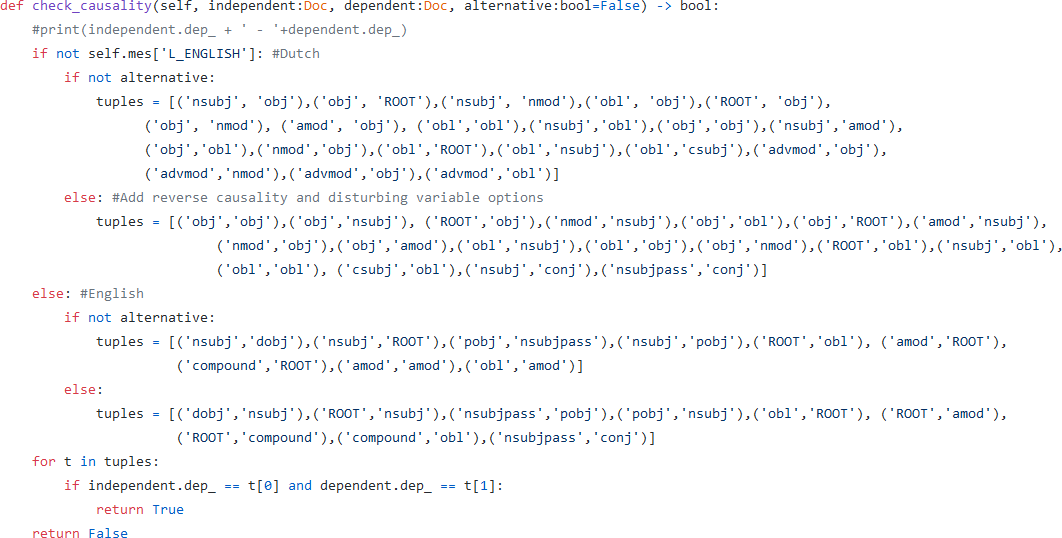


Figure 8: Check\_causality(), one of the most important help functions.

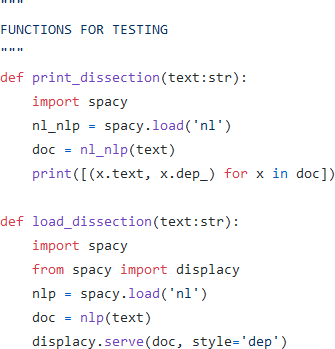


Figure 9: The test functions.