angrist 1.2(en)

Documentation and reference for the coder interface

Martin Wettstein

angrist, n ['æŋgrist] 1. Python script to generate a query form for relational data input in content analyses. It is especially useful for the application of hierarchical codebooks, as it allows data entry on different levels of analysis without raising the cognitive load of coders. 2. (from Sindarin: '*Iron cleaver*'): Legendary knive from the ballad 'The Lay of Leithian' by J.R.R. Tolkien. The young hero Beren uses this knive to descend to the lowest levels of hell in his quest to separate a precious gem from Morgoth's iron crown.

Zürich, May 2014

Current version available at:

http://www.tarlanc.ch/angrist/ANGRIST_Dokumentation.pdf

Inhalt

[..]and tried its hard edge, bitter-cold,
o'er which in Nogrod songs had rolled
of dwarvish armourers singing slow
to hammer-music long ago.
Iron as tender wood it clove
and mail as woof of loom it rove.
The claws of iron that held the gem,
it bit them through and sundered them;[..]

(J.R.R. Tolkien -The Lay of Leithian, 4143-4151)

1. Introduction

Angrist[1] is a highly adaptable script for displaying query inputs for quantitative content analysis. The special feature of this script is the ability to handle hierarchically structured codebooks and allows for the simultaneous coding on different levels of analysis within a given text. Accordingly, the data is exported in a relational database, pseudo-XML, or JSON-format, depending on the preferences and needs of the content analysis. By organizing the relational data input in linear and easy-to-accomplish tasks, data input is feasible even for highly complex codebooks without risking cognitive overload in coders.

Neither the data structure nor the complexity of the codebook has to be considered by the coder. Instead, the coder is confronted with a series of questions in natural language which may be answered from sets of previously defined items. The codes, storage, and interconnectedness of the variables are hidden from the coder to minimize cognitive load. For the task of answering the questions, Angrist provides a wide range of question-types, including lists, dropdown-menues, checkboxes, and rating-scales.

Since the coder interface runs as a stand-alone Python-script filtering of questions and pages is very easy and the tool is expandable with a wide variety of modules for data in- and output, as well as data processing, available for Python. One module specificially designed for the combination with Angrist even allows for the automated coding of certain variables by means of a learning algorithm which is automatically trained by manual data input.

1.1. Area of application

Angrist is especially useful in the analysis of texts using hierarchical codebooks. A wide variety of content analyses in different fields have made use of the coder interface so far, three prototypical ones of which are referred here:

- In the course of a project aiming for the quantitative analysis of public debates, articles have been divided to the statements of different actors, including the journalist himself. Each statement was again divided to single problem definitions, causes, evaluations, and treatments it contained. Accordingly, each text was analyzed on three levels of analysis (Text, Speaker, Statements). In total, more than 9000 texts have been analyzed, the largest of which containing more than 40 units of analysis.
- In the course of a project analyzing financial news, all financial developments which are part of the index SNP500 have been identified in articles within the finance-section of influential newspapers. For each development, all causes, consequences, and evaluations by actors have been coded.

[1] **A**djustable **N**on-commercial **G**adget for **R**elational data **I**nput in **S**equential **T**asks

- For the analysis of online comments, the comment threads to online news coverage have been investigated. Each comment was thereby divided to single references to other discussants, the topic of the article, or persons outside the discussion. A total of 250 articles with more than 11'000 comments and 16'000 references have been coded in this analysis. The coding of a single comment with all references thereby took less than three minutes.

All of these examples used three different levels of analysis. The script is expandable and would allow more levels of analysis. The impact of a higher complexity on the coding process and cognitive load of the coder has to be kept in mind, however. Also, as no analysis has ever used more than four levels of analysis in Angrist, the sound function of all features has to be tested before applying a codebook with more than four levels.

1.2. Information for Users

Angrist was designed as an Open Source academic Software and is subject to according regulations (http://www.opensource.org/docs/osd). All parts of this script may be copied, redistributed, changed, and adapted to personal tastes or needs.

In spite of intense care and rigorous testing of Angrist, programming errors may not be ruled out. This means that even correct use without changes to the program code may lead to damage to entered data or suspension of the function of this program; -although this is highly improbable. In the case of mishandling of this tool or changes in the program code, the probability of data loss increases.

The programmer declines any responsibility of the loss, damage or incorrect acquisition of data and is not liable to amend programming errors. In order to prevent damage to the program and data, the modification of the script should be limited to the part specific for the project. Modifications of built-in functions should only be done by well-trained persons.

The programmer is doing his best to correct any programming errors and to update the script on a regular base. The current version of the program may be found at:

http://www.ipmz.uzh.ch/Abteilungen/Medienpsychologie/Recource/Angrist.html

For questions and support, please do not hesitate to contact the programmer directly.

1.3. Technical detail

Angrist was written in python 2.7 (from version 1.1, the script also runs in Python 3.x) (www.python.org) and uses the TK-interface (library: tkinter) for displaying the GUI. The GUI is designed for Windows and does not run properly on Unix and Mac. For usage on these OS, please use a windows emulator.

The tool may display the text to be coded on screen. This display is currently limited to ASCII-Textfiles, the import of images, PDF-files and rich-text is planned for future versions, however. Please convert texts to be coded to ASCII-format prior to using them as text sources. Unicode encrypted documents may not be displayed properly.

2. Angrist GUI

The graphical user interface (GUI) of angrist is a single dialog window presenting the coder with a series of questions. The questions are arranged in subsequently presented pages which may be set up dynamically. On each page a maximum of three different questions may be asked, using different answering schemes such as dropdown-lists, lists, checkboxes, rating scales, or buttons.

Next to the question page, the coder may read the text to be coded and highlight portions of the text using predefined colors. All highlighted portions of the text may be read by the program and used as input for the content analysis.

The layout of the GUI is defined using the function `self.set_window()` and remains fixed for the whole analysis. Modifications of the layout may be done using this function and will affect all pages of the GUI. The standard design (see Figure 1) will display the pages on the left and the text to be coded on the right side of the dialog. The check-button, which is used to submit answers, is located at the bottom-left of each page. Since this design has been found to be irritating to some right-handed coders, the layout may be flipped using the setting `'Layout'`.

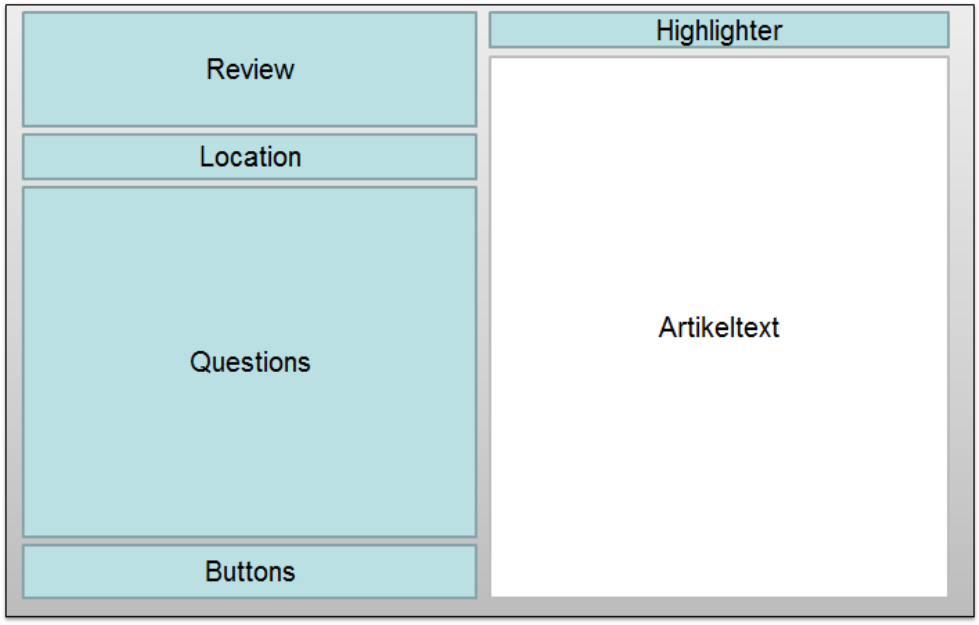


Figure 1: Standard-Layout of the Angrist GUI

Artikeltext

In this text area, the current text is displayed. The source for this text is an ASCII-formatted Text-file which is named after the ID of the current text.

Portions of this text may be highlighted for the purpose of navigation and data input. In most analyses, this feature is used to mark the units of analysis which should be coded in this text. Angrist may use the highlighted portions as input to guide the coder through the coding process.

If no text is available, this part of the dialog will not be used and the dialog only needs one half of the screen. The other half may be used for displaying texts or audiovisual documents using other applications.

Highlighter

When text is available, a menu of markers is displayed above. These markers may be used to highlight any portion of text with a given color. The colors, labels and the meaning of these buttons may be

defined using the setting `'Highlight_Buttons'`. In analyses using multiple levels of analysis, the markers may be used to highlight units of analysis within the text for later coding.

Review

At the top of each page, a frame is reserved for a review of previous data input. When visible, a list displays all units of analysis within a given level below the current level of analysis. This review-list may be called using the function `self.show_review()` and will be shown only on pages explicitly demanding it. On all other pages the list is disabled.

The elements within the review-list may be removed or edited by the coder if allowed. While removal is trivial to the program, editing of single items within the list requires some preparation. The function `self.edit_item()` specifies the pages which should be called when editing an item on a given level of analysis. The names of these pages have to be specified for each project.

Location

In the frame beneath the review-list, a text area is displayed. This area may be used to inform the coder on his current location in the coding process. The function `self.locate()` may be used to display information on each level of analysis.

Questions

The questions-frame is the central frame on each page. Within this frame, three questions and answering devices may be displayed to the coder. The questions may be called by using the `self.question_xx()`-functions. Each of these functions will display the question for a given variable at the specified position. Upon submitting the answers, this frame is cleaned and may be filled with new questions on the next page.

Please consider that there are question-types which need more space than others. Especially checkbox-lists or long series of rating-scales are consuming lot of the space available. While the GUI automatically reshapes to take in the whole page, this may lead to a window-size exceeding the size of notebook monitors. When using more than one Checkbox, Radiobutton or Rating-Question on one page, you might want to check the size of the page carefully.

Buttons

The bottom frame on each page contains up to four buttons with fixed purpose.

Check-Button: This button is linked to the function `self.submit()`. The submit-function is used to store current entries and set the program position for the next page. This button is vitally important on all pages except for pages which have other means to call the submit-function (e.g. a Buttons-question).

Abort-Button: This button is linked to the function `self.abort()`. The abort-function may be defined individually for each page where this button is available. Usually it is used to cancel the coding of a unit of analysis and return to the decision point.

Back-Button: This button is linked to the function `self.back()`. This function will set the program position of the last page visited and direct the coder there without storing current entries. Be aware that this function just goes back one page. For pages where the back-button should lead to entering another level of analysis, this has to be specified.

Break-Button: This button is linked to the function `self.pause()`. This function blanks out the page and start the timer for break-time. At the end of the coding, the break-time is stored to the data to keep track of net duration of the coders' work.

2.1. Question-Types

The GUI is able to present the coder with a wide variety of question types. These questions are called within the ASK-Function for all pages of the questionnaire. Each of the different question-types may be called using the appropriate `self.question_xx()`-Function. These functions take the codebook-variable and the position of the question as arguments and place the demanded question-type on the page. In this chapter, a short summary for all question-types is provided.

Dropdown

Dropdown questions present the coder with a list of options as defined in the codebook from which he may select the appropriate one. While each option has a code and a label in the codebook, only the label will be presented to the coder as an option. When calling a dropdown question, the first option in the list will be set as default value if no other default value has been specified.

When submitting the answer to a dropdown question, the code of the option will be stored to the data as a string variable. If this code happens to be '98', the answer is not accepted and the coder is prompted to select another option. Therefore, you may use the option 'Please select an option...' with code '98' as first option in the list. The coder will then be forced to make a choice to continue.

Dropdown questions are called using the function `self.question_dd(var,pos)` where `var` is the name of the variable as stored in the codebook and `pos` is an integer ranging from 1 to 3 indicating the position within the questions frame.

Text

Text questions ask for a manual input by the coder. This input may be entered to a single-lined textfile or to a large textbox. The values are stored as strings. Even if there are options for this variable in the codebook, these options will be ignored when calling a text question.

Text questions may be called using the function `self.question_txt(var,pos[,width])` for single-line text entries or `self.question_txt2(var,pos[,width[,height[,getsel]]])` for large textboxes. `var` is the name of the variable as stored in the codebook and `pos` is an integer ranging from 1 to 3 indicating the position within the questions frame. The width and height of textboxes may be changed using the corresponding arguments. The argument getsel is an integer with value 1 or 0 which indicates whether it should be possible to copy/paste a part of the displayed text by means of a button.

Checkbox

Checkbos questions may be answered by checking the options provided in the codebook. Up to 14 options may be defined in the codebook and will be displayed next to checkboxes.

When storing the answer to a checkbox-question, a dummy variable for each option is created which may take the values 0 (not selected) and 1 (selected).

Checkbox questions are called using the function `self.question_cb(var,pos)` where `var` is the name of the variable as stored in the codebook and `pos` is an integer ranging from 1 to 3 indicating the position within the questions frame.

Radiobuttons

Radiobutton questions may be used analogous to dropdown questions. Here, however, all options will be visible at the same time and the coder may select any one of the options as an answer.

When changing the value of a radiobutton question, the function self.rb_tamper() is called automatically. You may use this function to define an action which is to be taken upon selecting certain

values. For example, you may choose to ask for the name of the news agency if the coder selects the value 'News Agency' from a list of options.

Radiobutton questions are called using the function `self.question_rb(var,pos)` where `var` is the name of the variable as stored in the codebook and `pos` is an integer ranging from 1 to 3 indicating the position within the questions frame.

Buttons

Buttons question present the coder with up to four different choices which he may select by pressing the corresponding button. Each button may be linked to a function (either existing or custom) which will be called as an answer to the coders' decision.

No value is stored for buttons questions. They are usually employed at decision-points where the choice of a coder may lead to different courses in the coding process. Each event has to be defined when calling a buttons question.

Buttons questions are called using the function `self.question_bt(var,pos)` where `var` is the name of the variable as stored in the codebook and `pos` is an integer ranging from 1 to 3 indicating the position within the questions frame.

Rating

Rating questions present the coder with a list of items to be rated on a scale. By default, a 5-point likert scale, ranging from 'disagree' to 'agree' is used. The number of points, as well as their values and labels may be defined when calling a rating question.

When submitting the answers to a rating question, dummy variables for each option will be created which may take any of the values specified for the rating scale.

Rating questions are called using the function `self.question_rating(var,pos,scalelist, valuelist,defval)`, where `var` and `pos` again indicate variable and position. The parameter `scalelist` is a list containing all labels for the points of the rating scale. The parameter `valuelist` is a list containing the corresponding values. These lists have to have the same length. The number of points of the rating scale is equal to the length of these lists. If desired, a default value may be set using the parameter `defval`. If this value is off the scale (i.e.: not in the valuelist), no selection will be made on default.

Semantic Differential

As with rating scales, semantic differential (SD) questions prompt the coder to rate an item on a scale. The scale, however, spans for each item from one option to another one. This question type may be used to ask for a rating of a text from 'boring' to 'interesting' and from 'emotional' to 'sober'. The code of each option (as defined in the codebook) will be used as first option, the label will serve as second option for each item.

When submitting the answer to a semantic differential, a new variable is created for each item with the value assigned on the rating scale.

SD questions are called using the function `self.question_sd(var,pos,points,defval)`, where `var` and `pos` again indicate variable and position. The parameter `points` is an integer indicating the number of points which is to be used. If desired, a default value may be set using the parameter `defval`. If this value is off the scale (i.e. larger than points) no selection will be made on default.

List

List questions may be used to present the coder with long lists of options from which either one or several items may be selected by the coder. If desired, the list may be searchable by entering charcters to a textbox above the list. This option is recommended for lists with more than 20 items. In addition, the coder may be prompted to add list items to a smaller list instead of simultaneously selecting them. This is recommended for tasks where several items from a list have to be chosen.

When submitting the answer to a list question, a list-variable is created from the selection. If not processed any further, a string of this list-variable will be stored in the data.

List questions may be called using the functions `self.question_ls(var,list,multi)` for simple lists, `self.question_lseek(..)` for searchable lists and `self.question_ladd(..)` for searchable lists which ask for the export of items to a smaller list. Other than all other question types the list question may only be placed at position 1. The parameter `list` indicates the variable which contains the list options. This variable may be different from the variable containing the question. The parameter `multi` may take the values 0 and 1 and indicates whether multiple selections are allowed.

2.1.1. Help Text

For each variable, a short help text may be defined in the codebook. When calling a question, a small question mark is displayed at the right border of the page. When clicking on this question mark, a small Popup window appears, containing the help text for this variable.

2.1.2. Example

As an example, we define the variable 'Author' in the codebook and call it using all available question types. The definition is as follows:

```
[Author]
Who is the author of this text?
Please indicate the author
The author of a text refers to the person responsible for the content and
    presentation. In most cases the author is a journalist of the source in which
    the text has been published or a journalist of a news agency.[...]
1:Journalist of the Medium
2:News Agency
3:Reader / Audience
4:National political actor (Chose from Appendix B)
5:Other
6:Not identifiable
```

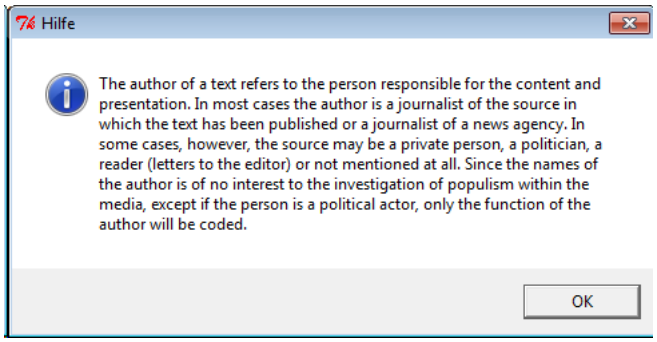
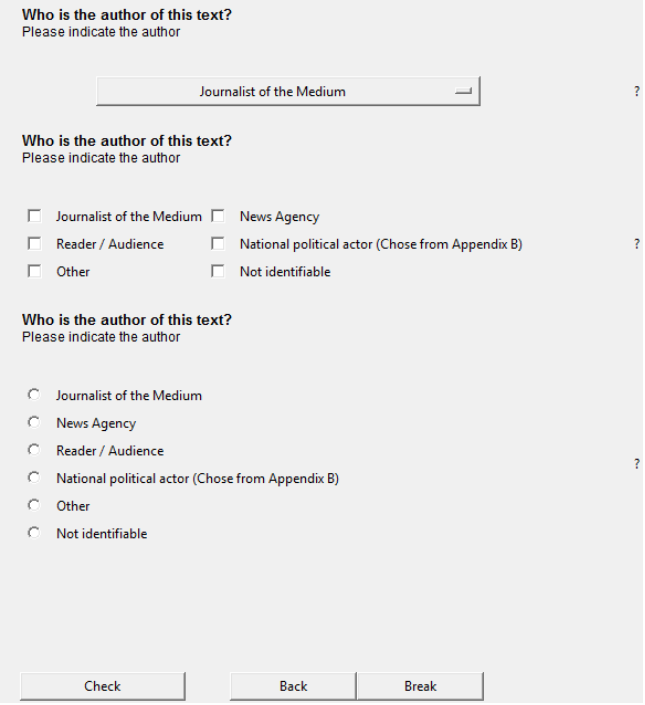
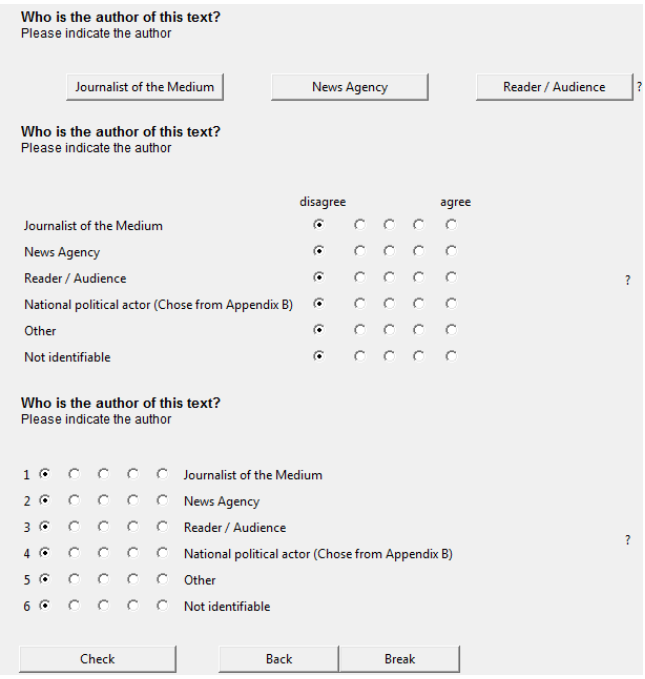


Figure 2: Help text which is displayed in a pop-up as soon as the coder clicks the question mark at the right-handed border of the page.

Page definition	Presentation

`self.buttons()`
`self.question_dd('Autor',1)`
`self.question_cb('Autor',2)`
`self.question_txt('Autor',3)`
	

`self.buttons()`
`self.question_bt('Author',1)`
`self.question_rating('Author',2)`
`self.question_sd('Author',3)`
	

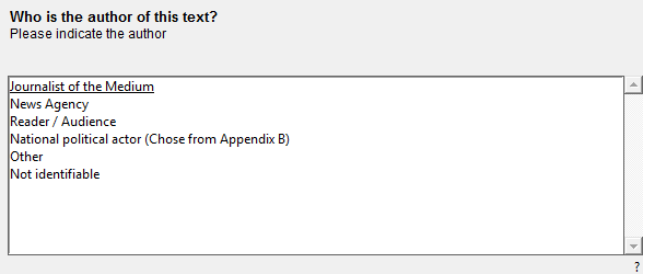
`self.buttons()` `self.question_ls('Author','Author')`	

Figure 3: Presentation of different question types for the same variable. As the variable was not designed for semantic differential questions, the example looks awkward for this question type. The coder has to rate the items on a scale ranging from the code of the item to its name.

2.2. Behind the Scenes

The script basically runs on one external file, two functions, and two global variables. All five constituents have to be adjusted to each other in order for the program to run smoothly and store all data correctly. In this chapter, a brief introduction to the constituents is provided. Please refer to the according chapters to find out more about each one.

The codebook (see chapter 3.2) is a file containing the questions to be asked and the answering options. Entries in the codebook may be changed independently from the rest of the program as long as the names of all variables remain unchanged. The ASK-Function contains a large IF...ELIF..ELSE block defining the questions to be asked on each page. All pages have an individual name and may contain the query for three variables within the codebook. The SUBMIT-Function contains a large IF...ELIF...ELSE-block which defines the actions to be taken upon submission of each page. Usually, the actions are storing, cleaning up and setting a new page. For each page, both an ASK- and a SUBMIT-handler have to be entered. The variable prog_pos contains the name of the current page. When running the ASK- or SUBMIT-function, only actions defined for the current value of prog_pos will be executed. Thus, the variable determines the exact position within the questionnaire. Finally, the variable dta_pos contains the current position within the data. This includes the names of the current level and the current unit of analysis as well as the levels and names of all units above the current unit of analysis. This variable determines where the program stores current entries. Figure 4: Illustration of the interconnectedness of all functions and variables in a short example. At two points, the entries made by the coder are influencing the coding process. First, by answering to Var1, Option A will lead to page p2 being displayed, Option B will lead the program to skip this page. Second, when answering F to Var3, an additional question (Var7) will be displayed on page p3.Figure 4 illustrates the coding process in a simple example.

When setting up Angrist, the codebook has to contain all variables used in the ASK-Function. Likewise, all pages have to be defined for the ASK and SUBMIT-function. If there are incongruences in the spelling of the names of pages and variables, the program will fail.

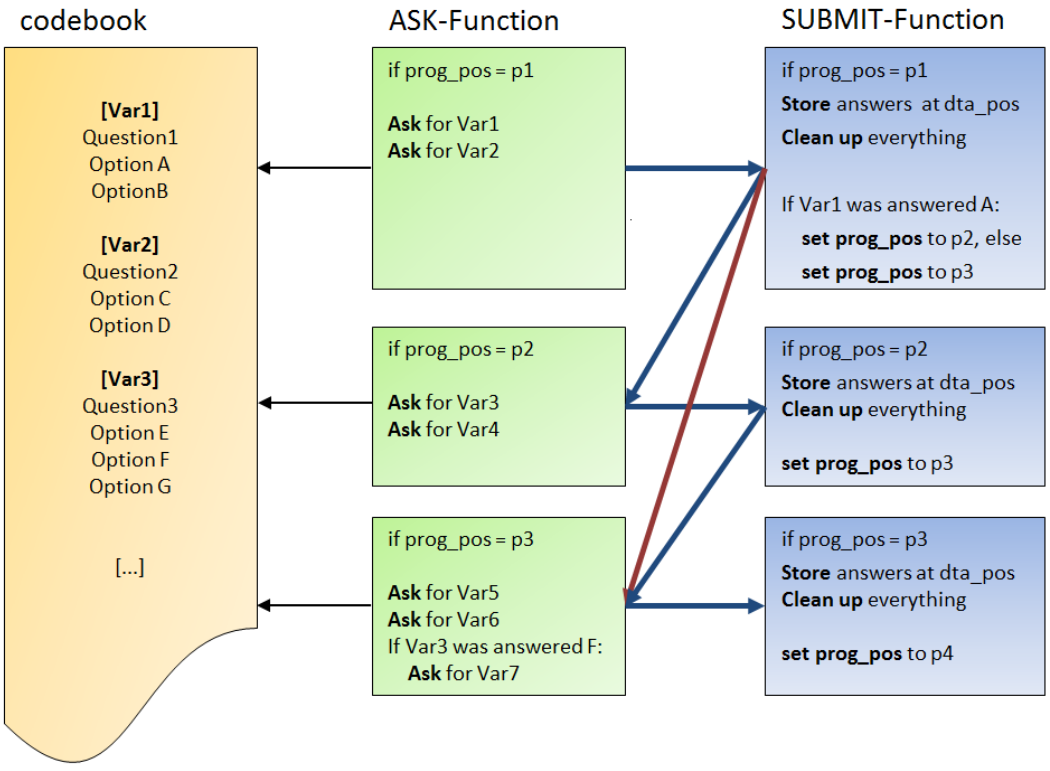


Figure 4: Illustration of the interconnectedness of all functions and variables in a short example. At two points, the entries made by the coder are influencing the coding process. First, by answering to Var1, Option A will lead to page p2 being displayed, Option B will lead the program to skip this page. Second, when answering F to Var3, an additional question (Var7) will be displayed on page p3.

3. Required files

Angrist was designed as a stand-alone python script. The program does only need built-in modules available in the standard configuration of Python 2.7. Accordingly, no additional scripts are needed for the execution of Angrist.

The minimal configuration for the script to work is the script (usually: `angrist.py`) itself and a codebook (usually: `a_codebook.ini`), which contains the questions and options for each variable. When both files are present in a directory and set up correctly, Angrist may be executed.

There are, however, two additional files which may be used to configure the tool, provide the script with additional information. An ini-file containing individual settings for each coder (usually: `a_settings.ini`) may be used to set the name of the coder, layout, presentation, and other parameters which vary between coders. A todo-list (usually `to_do.txt`) may be used to provide the coder with a list of text identifiers from which angrist may select the next text to be coded. Errors in the spelling of article identifiers may be prevented in this way.

In addition, a directory containing texts (usually: `Texts\`) may be used to store the texts in ACSII-format in order to be displayed during the analysis.

3.1. angrist.py

The file `angrist.py` is the central python script which initializes and controls the GUI for data input and stores the entered data in a specified format. The script has to be set up individually for each project and should run smoothly before starting the content analysis. During the phase of data acquisition, the script should not be modified any more.

The script was written in Python 2.7 but it also runs on Python 3. If no Python compiler is installed on a computer, an offline-version of python may be used from any portable device. The offline-version should contain the libraries *tkinter*, *os*, and *time*.

The modification of the script demands no advanced knowledge of Python. Basic operations, definition and calling of functions, as well as the application of methods to objects should be known, however. For a detailed reference of the programming language, see http://docs.python.org/tutorial/. For an introduction to the tkinter-library which is used to build the dialof for the GUI, see: http://www.pythonware.com/library/tkinter/introduction/

3.2. a_codebook.ini

The file `a_codebook.ini` is the main source of information for the script. The file contains standardized entries for each variable used in the analysis. Each entry contains a question to be asked, additional information for the coder, a help text, and a list of options to choose from.

The format of this file, which should be held in ASCII, is very simple and does not require any programming skills. For the correct execution of Angrist, the format of the codebook has to be followed strictly. The format is as follows:

```
[Name of the variable]
Question (1 line only)
Coder Information (1 line only)
Help Text (1 line only)
Code1:Item1
Code2:Item2
Code3:Item3
...
```

Each entry begins with the name of the variable in brackets [..] and ends with at least one empty line. Below the name of the variable there are three lines for text input. The first line contains the question to be asked. The second line contains coder information to be displayed beneath the question. The third line contains the text to be displayed if the coder requires a help text. Since each of these entries is limited to one line, no line-break is allowed in any of them. If you want to display line-breaks, use the number sign (`#`) to mark their location. When displaying text, these characters are equivalent to line-breaks in Angrist.

The list of options may contain any number of lines, each of which is used as an option. Each line is to be divided by a colon (`:`). The portion of the line left to the first colon is used as code of the option and will be stored to the data. The portion right to the first colon is the label of the option and will be displayed within the GUI. Both code and label may be any string of characters. If you omit the colon, label and code will both take the value of the whole item.

3.3. a_settings.ini

The file `a_settings.ini` contains individual information for each coder. This information usually includes the name of the coder, the preferred font and size, and the preferred layout. It may also contain a variable called `'Default-Values'` which contains defaults for some of the variables in the content analysis.

All information within this file will be included in the `settings` directory in Angrist. All settings which are defined in Angrist itself are overwritten if there is conflicting information in this settings-file. Thus this file may be used to change the set-up of Angrist for each coder individually. For example, it may be useful for the project leader to set the `'Debugging'` setting to 1 and for coders to set the `'Insecure'` setting to 1.

```
##Coder Informations

[Coder-Settings]
Individual Settings
All Options are stored in the settings-dictionary
All entries have the form Name:Value
Codierer:mw
Font:Arial
Layout:Righty
Fontsize:11
Insecure:0

[Default-Values]
Values to be retained
The default values are loaded at startup
All entries have the form Name of the Variable:Value
Source:3
```

3.4. Todo-List

The Todo-list is a list of filenames (with extension .txt) or article identifiers (without extension) on seperate lines which have to be coded. Angrist will take the first item from this list upon initialization and load the corresponding text from the text folder. Upon completion of a text, the entry is removed from the list.

Hint: The easy way to create a todo-list from a folder full of text-files: Create a textfile which contains the single line: `*.txt /b >..\to_do.txt` in the text folder. Change the extension of this file to `.BAT` and execute it by double-clicking it in windows. The command creates a todo-list in the parent directory containing all filenames on seperate lines.

3.5. Text folder

The text folder is a sub-directory (usually in the same directory as angrist.py) which contains all texts to be coded. The texts should be stored as ASCII Textfiles with the extension .txt. The name of the text folder may be changed using the settings-file.

3.6. Python Compiler

Since Angrist is not an executable but a python script, a python compiler is needed in order to run it. Python 2.7 is recommended but Angrist also runs on Python 3. If python may not be installed on the computer, an offline version (i.e. the Python directory from a computer where it was installed) may be used from any directory on the computer or from a portable device.

You may use the command `PATH\Python27\python.exe angrist.py` to execute Angrist with an offline version of the compiler. `PATH` is the path of the python directory.

4. Important variables

The script uses a large number of variables for internal storage and data management. For the programming and handling of the script, only few of them have to be known and understood. The most central variables are the `settings`-dictionary, the `storage`-dictionary and the position variables (`prog_pos` and `dta_pos`).

4.1. settings

`settings` is a global variable of the type dictionary[2] which contains all global settings of the script. There is a number of fixed keys within this dictionary which may be used to change the appearance and handling of the GUI and the program in the background. Next to these central settings, each project may define any number of additional settings as needed or desired.

The list below shows the most central settings which are required for built-in functions and should therefore not be overwritten by project specific variables:

Name	Variable Type	Purpose
Coder	String	The name of the coder. Used in any data output. Default: `'default'`
Font	String	The font used in text areas. Default: `'Arial'`
Fontsize	String	The size of the font of displayed texts. Default: `12`
Layout	String	Controls Right- or Left-handed layout. Default: `'Lefty'`
Curr_Page	List of Lists	Contains three lists with two items each that control the layout of each page. Do not manipulate!
Page_History	List of Strings	Contains a list of all pages visited in this coding session. Do not manipulate!
Input	List of SysVars	Contains dynamic variables with information on current input. Do not manipulate!
Path_Log	String	Contains the log-file of functions called in this coding session. Used for debugging.
Verb_Log	String	Contains the log-file of all text output. Used for debugging.
Verbose	String ('0'/'1'/'2')	Controls the verbosity of the program. If set to `0`, no text output is generated. If set to `1` only important events are written in the logfile. If set to `2`, the program is maximally verbose and comments all steps. Value `2` is used for debugging only.
Debugging	String ('0'/'1')	If set to `0` the GUI is set for coding. If set to `1`, the GUI may be used for debugging. In this mode all entries will be accepted and you may easily access `settings['Path_Log']` and `settings['Verb_Log']`
AEGLOS	String ('0'/'1')	If set to `1`, the aeglos module is used for automated content analysis in the background to set default values. Only use if aeglos has been installed and trained.
Insecure	String ('0'/'1')	If set to `1`, the coder may report insecurity regarding a decision. Only use in coder training.
Multi_Items	List of strings	Contains the list of variables which use dummy-variables in storage (checkbox-, sd-, and rating-questions). These variables will be

[2] In Python sind dictionaries eine Klasse von Variablen, in welchen Werte für unterschiedliche Variablen abgelegt sind. Sie können auch als indizierte Listen verstanden werden. Siehe http://docs.python.org/library/stdtypes.html#dict für weitere Informationen.

		provided extra space in the data.
Break	Integer	Contains the timestamp of the beginning of the current break. When not in break, the value is set to `0`.
Break_Time	Integer	Contains the total duration of breaks in this session.
Coding_Time	Integer	Contains the total duration of coding in this session.
Hotwords	Dictionary	Contains a list of keywords which may be highlighted in the text.
Auto_Highlight	String ('0'/'1')	Controls whether the Hotwords should be highlighted upon startup.
Todo	String	Defines the name of the todo-file. Default: `'to_do.txt'`. If the value is set to an empty string, no todo list is used.
Package_Todo	String	Defines the name of the todo-file containing the names of folders each containing another todo-list and a set of texts. Useful for assigning packages of articles to the coders. If set to an empty string no package todo list is used. Default: `''`
Codebook	String	Defines the name of the codebook file. Default: `'a_codebook.ini'`. If set to an empty string, the file `'a_codebook.ini'` will be used as codebook file.
Settings	String	Defines the name of the settings file. Default: `'a_settings.ini'`. If set to an empty string, the file `'a_settings.ini'` will be used as settings file.
Text_Folder	String	Defines the folder which contains the texts. Default: `'Texts\\'` (mark that backslashes must be escaped!)
Out_Track	String	Defines the name of the tracking-file. All steps in the coding process are logged to this file. Useful for debugging. If set to an empty string, no tracking is used. Default: `''`
Out_Tree	String	Defines the name of the hierachical output of all data. If set to an empty string, no hierarchical output is used. Default: `'trees.txt'`
Out_JSON	String	Defines the name of the JSON ouput file. If set to an empty string, no JSON output is used. Default: `'json_dump.txt'`
Out_Tmp	String	Defines the name of the temporary storage. If the coding process gets terminated, all data is stored in this file. Currently there is no possibility to recover the coding automatically. Angrist 1.2 will be able to do so.
First_Page	String	Defines the page from which the coding process should be started.

The default value for each of these settings is set at the bottom of the script. If you wish to change the default values you may either change them at the bottom of the script (for all coders and once per session) within the function `self.fuellen()` (for all coders, set back to default for each text), or by using the settings file (see chapter 3.3).

If you are not sure whether a parameter has been set in the settings dictionary you may check using the function `available(setting)` where `setting` is a string variable with the name of the setting you wish to find. If there is a valid value in the settings dictionary for this key, the function returns `True`. If there is no valid setting, the function returns `False`.

4.2. Storage

`storage` is a global variable of the type dictionary. It contains all stored data for the current text. If there are multiple levels of analysis, the dictionary contains dictionaries for each unit of analysis. The format of values stored in this dictionary depend on the question type used for entering the values.

- For dropdown and radiobutton questions, storage contains a tuple with two elements (Label,Code) as string variables as value.
- For text questions, storage contains a string variable as value.
- For checkbox, rating, and sd questions, storage contains a dictionary with all options as keys and the entered values as values. For rating scales, the values may be defined manually. For checkbox and sd questions, the values are integers.
- For List questions, storage contains a tuple of lists ([Label1,Label2...],[Code1, Code2..]) which contains all labels and codes selected. If no multiple selection is allowed, the lists only contain one element.

Below you see a very brief example of the coding of a text using two levels of analysis (Text and Speaker) with some variables to code on each level. There are two units of analysis on the level of Speaker (S1 and S2), each with their own storage dictionary. On the level of text, the ID and length of the text were coded using a text question. The variables Genre and Author were coded using a dropdown menu. Framing was coded using a checkbox question with four options. On the level of speaker, three variables (ID, Position and Style) were coded using dropdown questions.

On each level there are two reserved variables which are coded automatically. `#TN` is the trivial name of the current unit of analysis. It is set in the function `self.level_down()` and is used for the review of coded units and informations on the current location. Its value may be changed at any time. The second reserved variable is `#TS`, the time stamp of the current unit of analysis. It marks the second the coder started coding this unit.

```
{
#TS:1381235730.75
ID:'Text1'
Genre:('Report','1')
Author:('Journalist','1')
Length:'300'
Speaker: {
    S1: {
        #TN:'Barrack Obama'
        #TS:1381235916.45
        ID:('Barrack Obama','1034')
        Presentation:('Positive','1')
        Voice:('Yes, direct quotation','2')}
    S2: {
        #TN:'Michelle Bachmann'
        #TS:1381235950.763
        ID:('Michelle Bachmann','1042')
        Presentation:('Negative','-1')
        Voice:('No','0')}
Framing: {
    Conflict:1
    Horse-Race:0
    Stratgy:0
    Attack:1}
}
```

Any entry in the storage may be assessed at any time in the program. If you wish to extract the code of the position of the first speaker, for example, you might call:

```
position = storage['Speaker']['S1']['Position'][1]
```

The value `'1'` would then be stored in the variable `position`.

When you are at a lower level of analysis and require a value coded within this unit of analysis, it may be doleful to enter the whole path to the current position manually. There is a shortcut you might use in this situation: If you call the function `curr()`, it returns the current sub-directory of storage you are editing. So, if you are still coding the first speaker and wish to know his position, you may call it using the command:

```
position = curr()['Position'][1]
```

Again, the value `'1'` is returned. This function is especially useful for analyses with more than two levels of analysis.

4.3. Codebook

codebook is a global variable of the type dictionary. It contains all variables of the codebook (usually a_codebuch.ini) as keys and the entries as value. The codebook dictionary is used in the presentation of question types and in the storage of variables. You may change the contents of the codebook at any time of the coding process.

The format of each entry in the codebook dictionary is a list containing five elements. Element 0 is the question asked to the coder as a string variable ending with a line-break. Element 1 is the coder information as a string variable. Element 2 is the list of labels for the options specified. Element 3 is the list of codes for the options specified. Element 4 is the help text offered to the coders.

Accordingly, the codebook entry for the example variable 'Author' (see Chapter 2.1.2) would be:

```
['Who is the author of this text?\n','Please indicate the author',['Journalist
      of the Medium','News Agency','Reader / Audience','National political
      actor','Other','Not identifiable'],['1','2','3','4','5','6'],'The author
      of a text refers to the person respo...']
```

When changing the entries of the codebook, these changes will persist for the whole coding session. They will, however, not be stored to the codebook file at the end of the text. The codebook is reloaded for each text in the coding session.

4.4. def_val

`def_val` is a global dictionary containing default values for certain variables. The keys of this dictionary are the names of the variables, the values are the values which should be set as default. You may set default values at any time of the coding process. If default values are set for a variable, the corresponding option is automatically selected as soon as the question for this variable is asked.

You may, for example, set the default value for the author of the text to 'Journalist', which has the code '1'. For this purpose you may use any of the following notations. All of them will work:

```
def_val['Author']='Journalist'
def_val['Author']='1'
def_val['Author']=('Journalist','1')
```

Setting default values is especially useful when using text- or data-mining techniques to predict the values of variables. It is also used to present the coder with the predictions of the automated content analysis using Aeglos (see Chapter **Fehler! Verweisquelle konnte nicht gefunden werden.**).

4.5. prog_pos

`prog_pos` is a global variable of the type string. This variable contains the name of the current page the coder is presented with. The value of prog_pos is usually set and changed in the Submit-function after storing the values and before calling the Ask-function again.

Hint: It is strongly recommended to use informative names for the pages of your questionnaire. Names such as `'page1'`, `'page2'`, `'page2.2'` will make setting up of your program a nuisance. If you use `'author'`, `'topic'`, `'framing'` instead you will always know where you are in your script.

4.6. dta_pos

`dta_pos` is a global variable of the type list of strings. The value of this variable specifies the position within the data the coder is currently storing values to. The list has a fixed format:

```
['Level1','Unit1','Level2','Unit2','Level3','Unit3'...]
```

This list always contains at least four items. If the coder is at the top level of analysis, the value of the variable is `['-','-','-','-']`. When descending to lower levels, the dashes are replaced by the respective values. When coding the first speaker in the example above, the data position has the value `['Speaker','S1','-','-']`.

The value of the dta_pos variable is changed automatically by the `level_down()` and `level_up()` functions. You may change the value manually at any time. When doing this, be careful to match the entries in `dta_pos` to the currently existing dictionaries in `storage`.

5. Setting up Angrist

To demonstrate the process of defining a new project in Angrist, a very superficial and small example project is presented here. Parts of this example have been used in previous chapters. Here, we will set up the tool as a whole.

The aim of this example is to investigate the presentation of political actors in an American newspaper. For this purpose, the Genre of each article, its length and framing have to be recorded. In addition, all political actors appearing within the article shall be recorded and their presentation shall be assessed using two categories: Quotations and tone of presentation.

5.1. Making a plan

The first step when defining a new project should consist in a detailed plan of the pages of the questionnaire which are displayed in the tool. This plan may be used to subsequently set up the codebook, the Ask-Function, the Submit-Function, the Back-Function and other project specific parameters.

The plan should contain the following information:

- The names of the pages
- The names of the variables
- Question types
- Changes of the level of analysis

Table 1 shows an example of such a plan containing all relevant information.

Page	**Questions: Variable and type**	**Actions upon submission**
article properties	Author (dropdown) Genre (dropdown) Length (text 1 line)	
actor selection	Act_ID (dropdown)	If an actor is selected, change the level of analysis to Actor. Else go to page: framing
actor properties	Presentation Voice	Go back to actor selection and change to the top level of analysis
framing	Framing (rating scale 2 points)	
otherart	Otherart (dropdown)	If yes, restart angrist, if no, end angrist.

Table 1: Detailed plan for the example analysis

5.2. Definition of the codebook

First and foremost, the variables needed in this analysis have to be defined in a codebook. For this purpose, an empty textfile is created and stored as `a_codebook.ini`. This textfile is then filled with entries for the variables specified in the plan. As described in chapter , each entry begins with the name of the variable in brackets which is followed by three lines of information and a list of options.

For the variables used in this examples, the codebook would look like that:

```
############### Article Level

 [Author]
Who is the author of this text?
Please indicate the author
The author of a text refers to the person responsible for the content and presentation. In
       most cases the author is a journalist of the source in which the text has been
       published or a journalist of a news agency. In some cases, however, the source may
       be a private person, a politician, a reader (letters to the editor) or not
       mentioned at all. Since the names of the author is of no interest to the
       investigation of populism within the media, except if the person is a political
       actor, only the function of the author will be coded.
1:Journalist of the Medium
```

```
2:News Agency
3:Reader / Audience
4:National political actor
5:Other
6:Not identifiable

[Genre]
Type of the text
Which genre/category does this text belong to?
News story#Fact-oriented type of story, factual news report, report of events etc., of
       what has happened [when, where, who, what, why?], e.g., party meeting, report on
       recent events etc. Probably the most frequent type of news
       story.##Editorial/Column/Commentary/Letter to the editor#Most subjective kind of a
       news item in which authors give their personal interpretations and opinions:
       #Editorial: Typically explicitly marked as editorial, opinion-piece, an article of
       its own, clearly defined to give evaluations, typically on same page within
       newspaper each time. It has to be formally distinct from the rest of the page. It
       clearly expresses a standpoint of the author/editor who again speaks for his
       newspaper. #Column: clearly marked as special column, distinct from regular
       coverage, most likely always at the same place within news-paper, reoccurring item
       on a regular basis as fixed part of newspaper coverage, can be written in very
       personal style. #Commentary: Often not written by a journalist but by an external
       source such as an expert, politician etc.; often explicitly marked as „commentary“,
       e.g. by guest author. Letter to the editor: written by a reader; referring to
       previous articles published in the newspaper and representing the personal opinion
       of the reader; mostly explicitly marked as letter to the editor and placed on a
       special page within the newspaper.##Interview#News item consisting of questions and
       answers shown or printed in direct quotes between interviewer and interviewee or
       between several discussants. Note: There have to be at least two interview
       questions (often in bold or italic) to justify coding an interview! Interview
       sections which are part of a „reportage“ or another more subjective report are not
       meant here.##Background report/Reportage/Portrait/Feature#More subjective reports
       in different forms: #Feature article: Vivid report of a correspondent, named as the
       author of the article. #A „reportage“ describes individual experience of the
       author; often explicitly marked as „reportage“.#Background story: often longer
       article, not only factual reporting, looking behind the scenes, analytical, in-
       depth – not only descriptive, often explicitly marked as „analysis“, etc. Magazine
       style report: Type of news story in which elements of factual news reports and
       subjective elements intermingle. But: The journalist does not have to be on
       location, describing individual experience as in a reportage/background
       story.#Portrait: Portrait of, e.g., a person, group, institution, organization –
       and nothing more than that. Otherwise it may be a news story or a reportage (see
       above).
1:News Story
2:Editorial/Column/Commentary/Letter to the Editor
3:Interview
4:Background Report/Reportage/Portrait/Feature

[Length]
Length of the text
Please enter the number of words below.
The number of words may be found at the top of the text next to the word 'LENGTH:'

[Framing]
How is the text framed?
Please check the presence and absence of these frames
Conflict-Framing:#There is a conflict between two or more actors. There is reproach, a
       winner, or a loser.##Horse-Race Framing:#Candidates are presented in a sports-like
       fashion and their chances to win or to lose are caluclated. There are points,
       percentages, or polls to support the prediction.##Strategy-Framing:#Politicians are
       portrayed as schemers making plans, intrigues, tactics, or strategies to reach
       their goals. Politics in general is portrayed as a strategy game.##Attack-
       Framing:#Direct attacks of actors against others are portrayed or the journalist is
       directly attacking a politician.
Conflict:Conflict-Framing
Horse-Race:Horse Race-Framing
Strategy:Strategy-Framing
Attack:Attack-Framing


[Otherart]
Do you want to code another article?
All entries you made have been stored. You may now choose to continue with another
       article.
-
1:Yes
2:No
```

```
############### Actor Level

[Act_ID]
Which actor is it?
Please select the appropriate actor from the list below.
You may enter some characters of the name of this actor to restrict the list.
x:No more actors
1:Boehner, John (Rep., R.)
2:Cruz, Ted (Sen. R.)
3:Gray, Vincent C. (Mayor Washington DC)
4:Holmes Norton, Eleanor (Del. D.)
5:Issa, Darrell (Rep. R.)
6:Needham, Michael
7:Obama, Barrack (Pres. D.)
8:Reid, Harry (Sen. D.)
9:other actor

[Presentation]
How is the actor presented?
Please select
The presentation refers both to verbal presentation and pictures.
1:Positive
0:Ambiguous/Neutral
-1:Negative

[Voice]
Is the actor quoted in this article?
Please select
A direct quote is always put in quotation marks. Indirect quotes may be recognized by the
0:No
1:Yes, indirect quote
2:Yes, direct quote
```

Please do not bother to take the content of this codebook seriously. It was created for purely formal reasons and does not comply with standards of quantitative content analysis.

5.3. fuellen()

With the codebook defined we may now turn to setting up the script angrist.py. For this purpose, a series of functions has to be defined and customized.

The first of these functions is `fuellen()` which sets initial values for the settings, storage and codebook directories and defines global variables. While most of this function should not be modified, you might want to change some of the settings for this content analysis.

Since it would be nice to automatically highlight the names of the actors we code in this analysis, we might insert the following lines:

```
settings['Hotwords']={'us':['Boehner','Cruz','Gray','Holmes','Norton','Issa',
                            'Needham','Obama','Reid']}
settings['Country']='us'
settings['Auto_Highlight']='1'
```

These lines set the country of the content analysis to `'us'` and cause the GUI to highlight a list of words upon loading. The words are case sensitive but may occur at any place within a word.

In addition to these settings we want to start the content analysis on page 'article properties' and enable the coder to highlight found actors using a yellow marker. Finally, one of the variables is entered by means of a rating question and will need more space in the data and we want to reset the list of actors for each text.

```
settings['Highlight_Buttons'] = [['Actor','Act','#ffff66']]
settings['First_Page'] = 'article properties'

settings['Multi_Items'] = ['Framing']
if 'Actor' in settings.keys():
    del settings['Actor'] ##Reset upon loading new text
```

Except from these additions, the `fuellen()`-function may be left unchanged for this project.

5.4. ask()

The Ask-Function contains all information for the display of questions. For each page, a series of commands may be specified, which call questions and the buttons at the bottom of the page. The commands are embedded in a large `if..elif..else`-block in which a command block is defined for each possible value of the variable `prog_pos`.

In this project we shall need five of these blocks which are called as specified in the plan above. In each block the questions for the variables on this page may be called using `self.question`-functions. In addition, a review of previously coded actors may be displayed at the page `'actor selection'`.

```
if prog_pos == 'article properties': #First page if no ID was found in any to-do
       list
    self.question_dd('Author',1)
    self.question_dd('Genre',2)
    self.question_txt('Length',3)

elif prog_pos == 'actor selection':
    self.show_review('Actor',1)
    self.question_dd('Act_ID',1)

elif prog_pos == 'actor properties':
    self.question_dd('Presentation',1)
    self.question_dd('Voice',2)

elif prog_pos == 'framing':
    self.question_rating('Framing',1,['present','absent'],['1','0'],defval='0')

elif prog_pos == 'otherart':
    self.buttons(0,0,0,1)
    self.ausblenden()

    check_todo()

    self.question_bt('Otherart',1)
    self.f_questions.bu1_1["command"] = self.submit
    self.f_questions.bu1_2["command"] = self.abort
    self.f_questions.bu1_1.bind('<Return>',self.submit)
    self.f_questions.bu1_1.focus()
```

Since the text is assessed completely by the time the coder reaches the page 'otherart', the current ID may be checked off on the todo list. To do so, the function `check_todo()` is called. This function removes the current ID from the todo list. Also, at the end of a text, the text display may be removed. This is achieved by calling `self.ausblenden()`. Afterwards, the question whether another article should be coded, is displayed to the coder. Since the question is a buttons question, the check button is suppressed on this page by means of the function `self.buttons()`.

5.4.1. Data storage

The code above would ask the coder for all entries but it would not store any information to external files. All information entered by the coder would be lost after finishing the text. Data may be stored using built-in functions which may store the data as tree hierarchy, as JSON output or as relational tables.

For all three ways of data storage, there is a function. Since the functions need a setting defining the name of the output-file it pays only to call them if the setting is available. You may check availability of any setting by calling `available(setting)`.

```
baum_export()

self.export_data([],['#TS','Author','Genre',
                     'Length','Framing'],'_Text.txt')
```

```
self.export_data(['Actor'],['#TS','Act_ID','Presentation',
                            'Voice'],'_Actor.txt')
```

These commands produce output in four different files. First, a tree representation of the storage dictionary is stored. Depending on the values of `settings['Out_Tree']` and `settings['Out_JSON']` an indented tree or a JSON-Output or both is written to a file. Second, all data on text level is exported to a table containing the columns `'#TS'`, `'Author'`, `'Genre'`, `'Length'`, and four dummy variables for the variable `'Framing'`. Finally, for all elements on level `'Actor'`, a case is exported to a table containing the columns `'#TS'`, `'Act_ID'`, `'Presentation'`, and `'Voice'`. In addition, each case will contain information on the coder, the text ID and the duration of the coding.

5.4.2. Calling functions using button questions

In this example there is only one button question at the end of the questionnaire. Variable `Otherart` may be answered with yes or no, leading to different paths in the coding process. For these decision posints, button questions may be used.

Button questions generate up to four buttons from options provided in the codebook. In this case, a button named 'Yes' and a button named 'No' is generated. These buttons are void of any function and have to be defined in order to work.

There are two different possibilities for linking these buttons to functions. First, you may bind the buttons to different existing or custom functions in which commands for the current program position are defined. This is what is done in the example above by linking the first button to self.submit() and the second button to self.abort().

Another way to bind buttons to functions is binding all buttons to the submit function with unique parameters. The function submit() takes one argument if necessary. This argument may be used to identify the button pressed. To submit an argument to the Submit-function, the CMD-class has to be used as no parameters are usually allowed when binding a command to a button:

```
self.question_bt('Otherart',1)
self.f_questions.bu1_1['command'] = CMD(self.submit,1)
self.f_questions.bu1_2['command'] = CMD(self.submit,2)
```

This code links both buttons to the Submit-function but it will set the value of the parameter `overspill` to `1` or `2`. This value may be used in deciding which action to take.

5.5. submit()

The Submit-function is called when pressing the check-button. It handles the internal storage of items, the cleaning up of pages and the redirection to subsequent pages. In the plan, three exceptional actions have been noted. When storing the entry of the page `'actor selection'`, the level of analysis has to be changed to `'Actor'`. Then storing the entries of the page `'actor properties'`, the level of analysis has to be set back to the top level. When storing the entry of `'otherart'` (in the definition of this page, the first button of the buttons question has been linked to `self.submit()`, so the coder wants to code another text), the GUI is reset by calling `self.fuellen()`.

The Submit-function always checks the entries before doing anything else. This is done by calling the function `self.check_entries()`. If the check fails, an error message will be produced and the entries will not be stored or processed.

5.5.1. Storing values

In the Submit-function, entries by the coder may be stored to the internal storage dictionary. To do so, you may use the function `self.store_var(var)` or `self.store_var_all()`. Both functions store the

entered value of one or all variables on the page to the respective key in the storage dictionary at the data position indicated by `dta_pos`. If you want to store the values to some other key within this dictionary you may do so manually.

For the page 'text properties', all three versions of storage commands shown below do exactly the same thing:

```
if prog_pos == 'article properties':
    self.store_var_all()

if prog_pos == 'article properties':
    self.store_var('Genre')
    self.store_var('Author')
    self.store_var('Length')

if prog_pos == 'article properties':
    storage['Genre'] = self.store_var('Genre',store=0)
    storage['Author'] = self.store_var('Author',store=0)
    storage['Length'] = self.store_var('Length',store=0)
```

The parameter store=0 in the function call of self.store_var() causes the value not to be stored automatically but to be returned in order to store it manually. In this case, the result is the same. You may, however, choose to look at a value without storing it yet. In these cases, the option is used.

5.5.2. Changing the level of analysis

There are different ways to change the level of analysis. The most simple one consists in calling the function `self.level_down(var,level)`, where `var` denominates the variable containing information on the identity of the unit and `level` denominates the level on which the unit of analysis is coded. Conversely, the function `self.level_up()` may be called to return to the level above the current one.

There is another way to change the level of analysis which is not used in this example. There, the coder is prompted to highlight all units of analysis within the text. The highlighted portions of the text are counted and will be presented to the coder who may then code them subsequently. To use this feature, an additional page has to be included and the coder has to have a marker defined in `settings['Highlight_Buttons']`.

On a first page, the coder is prompted to mark all units of analysis by using the question `self.question_mark_units(var,Level)`. The highlighted portions may be stored in the Submit-function by calling `self.unit_confirm(Level)`. Subsequently, by using the question function `self.question_sel_units(var,Level)`, in the Ask-function, the coder is prompted to select the next unit of analysis. Finally, the selection may be used to change the level of analysis by calling the function `self.unit_select(Level)` in the Submit-function. For all functions, the value Level refers to the level of analysis which shall be used to store the units. `var` denominates the codebook variable containing the question- and help-text for the questions.

In this example the handshake between these functions would look like this:

Ask-Function:

```
elif prog_pos == 'mark actors':
    self.buttons(0,0,0,0)
    self.question_mark_units('Mark_Act','Actor')

elif prog_pos == 'select actor':
    self.buttons(1,0,0,1)
    self.clean_all_tags() #clean up highlights
    self.show_review('Actor',1) #show previously coded units
    self.question_sel_units('Sel_Act','Actor')
```

Submit-Function:

```
elif prog_pos == 'mark actors':
    self.clean_up_all()
    self.unit_confirm('Actor')
    prog_pos = 'select actor'
    self.ask()

elif prog_pos == 'select actor':
    self.clean_all_tags()
    self.unit_select('Actor')
    self.clean_up_all()
    self.ask()
```

Evidently, two additional variables have to be defined in the codebook for this to work.

When creating a new unit of analysis using this method, the position of the highlighted text, its content and the content of the whole paragraph are stored in a dictionary in settings. There, too, a value is stored indicating whether this unit of analysis has been coded already or has to be shown to the coder again. In addition, the type of the marker and a label for list display are stored.

Key	Wert
Start	Position of the first character of the selection in the text
End	Position of the last character of the selection in the text
Fulltext	Text of the whole paragraph
Wording	Text of the highlighted portion of text
Done	Integer with value 0 for uncoded units and value 1 for coded units of analysis.
Label	Label of the unit of analysis to be displayed in lists. Usually shorter than 40 characters.
Typ	Type of the marker used to highlight this unit of analysis as defined in `settings['Higlight_Buttons']`

Table 2: Keys and content of the dictionary created at settings[Level][Unit] for each highlighted portion of the text.

5.5.3. Cleaning up the page

The page may be cleaned from all questions using the function `self.clean_all()`. If only one question is to be removed while the other ones remain in place, you may also call `self.clean(var)` where `var` denominates the variable behind the question to be removed.

5.5.4. Complete submit function

Taken together, the submit function for the current program looks like that:

```
accept_entry = self.check_entries()
if accept_entry == 1:
    if prog_pos == 'article properties':
        self.store_var_all()
        self.clean_up_all()
        prog_pos = 'actor selection'
        self.ask()

    elif prog_pos == 'actor selection':
        act = self.store_var('Act_ID',store=0)[1]
        if act == 'x':
            self.clean_up_all()
            prog_pos = 'framing'
            self.ask()
        else:
            if self.level_down('Act_ID','Actor'):
                self.store_var_all()
                self.clean_up_all()
                prog_pos = 'actor properties'
                self.ask()

    elif prog_pos == 'actor properties':
        self.store_var_all()
        self.clean_up_all()
        self.level_up()
```

```
        prog_pos = 'actor selection'
        self.ask()

    elif prog_pos == 'framing':
        self.store_var_all()
        self.clean_up_all()
        prog_pos = 'otherart'
        self.ask()

    elif prog_pos == 'otherart':
        self.fuellen()

    else:
        if settings['Verbose'] == '1':
            verb("Error: Position '"+prog_pos+"' is not defined for SUBMIT")
else:
    verb('Invalid Entries')
```

Within the command blocks, the storage function has to be called before the cleaning function. Upon cleaning, most entries are removed from the cache and the entries may not be stored anymore.

All commands within the Submit-function have to end with calling `self.ask()` again. If this function is not called, the GUI stalls and does not proceed with data input. If the value of `prog_pos` is not changed before calling the Ask-function, the GUI enters an endless loop.

On page `'actor selection'` the value of variable `'Act_ID'` is used to decide which path to take. If the selection is `'No more actors'` which has code `'x'`, the next page is `'framing'`. For all other values of `'Act_ID'` the level of analysis is changed. This is done using an `if..else` statement.

At some points of the Submit-function, the function `verb()` is called. This function takes one string as argument and writes the string to a log. The log may be seen when debugging but remains hidden from the coder.

5.6. abort()

The function self.abort() will be called if the abort button is pressed by the coder or if a buttons question links a button to this function. In this example, the abort button is not displayed on any page. There is, however a link to the function in the definition of the buttons question on page `'otherart'`. This means that the Abort-function has to be defined for this page:

```
        if prog_pos == ['otherart']:
            self.clean_up_all()
            prog_pos = 'ende'
            self.ask()
```

In this case, the page is cleaned up and the coder is redirected to the page 'ende' which informs him that all data has been stored and the window may be closed. Such a definition has to be done for any page where a button calls the function `self.abort()`.

5.7. back()

The function self.back() is called whenever the coder clicks the Back-button at the bottom of the page. This button is displayed by default and has to be disabled manually for pages where it makes no sense.

The Back-function usually sets the value of prog_pos to the last page visited, cleans up the page and calls self.ask() again. If any other actions should be taken when pressing 'Back' on a page, commands have to be defined manually.

In this example, special precautions have only to be taken for one page. When going back from the page 'actor properties', the level of analysis has to be set back to the top-level. This may be done using the following lines:

```
if prog_pos == ['actor properties']:
    del storage[dta_pos[0]][dta_pos[1]]
    self.level_up()
    prog_pos = 'actor selection'
```

The function `self.ask()` will be called at any rate so it does not have to be called in the if..else block of this function.

5.8. rb_tamper()

The function `self.rb_tamper()` is called whenever the value of a radiobuttons question is changed by the coder. The function may be used to change the appearance of the current page according to the entry of the coder.

As in all other functions in angrist, `self.rb_tamper()` contains an `if..elif..else`-block in which you may define commands for any page. Since no radiobutton is used in the example above, another example is shown here to demonstrate the definition of this function.

You may chose to ask for the exact news agency if the coder selects `'News Agency'` (code 2) as author of the text. If the variable 'Author' is answered by radiobuttons. Imagine there is another variable 'NewsAgency' which contains all news agencies as options:

```
if prog_pos=='article_properties':
    self.clean_up('dd',3)
    if storage['Author'][1] == '2':
        self.question_dd('NewsAgency',3)
```

These lines first remove the question at position 3 (if available). Then, if the value of `'Author'` is `'2'`, variable `'NewsAgency'` is asked using a dropdown question.

With the definition of these functions, the program runs perfectly well. The whole example as a working angrist project may be downloaded from the resources page at:

http://www.ipmz.uzh.ch/Abteilungen/Medienpsychologie/Recource/Angrist.html

6. Extension: Aeglos

As a python script, Angrist may be extended using a wide range of packeages for statistical processing, natrual language recognition, or data management which are available for python today. There is, however, one package which is designed explictly for the use with Angrist and may be used for semi-automated content analysis.

Semi-automated content analysis (SACA) is a method in which techniques for automated content analysis are linked to manual data entry with the purpose of enhancing the efficiency of the coders (Wettstein, In Press). A central element of SACA is the prediction of values the coder is likely to enter in order to preselect them in the query form. This has been found to have two different effects on the coder. First, the coder does not have to bother clicking the obvious choice from a dropdown which saves time and trouble in the content analysis. Second, the task of the coder changes from assessing the text on his own to checking and correcting the program's analysis. In cases where the program does not select the option which would be preferred by the coder, he reacts with increased elaboration of the text and pays higher attention to the semantic meaning of the text (Wettstein, in Press).

The Angrist Extension Generating Listselections for Optimal Speed (Aeglos) is a module enabling Angrist to learn from previous decisions of the coder and calculate the probability for the selection of any option for any variable in the codebook for unknown texts. The technique used for these predictions is a Naive Bayes Classifier (cf. Manning/Schütze...) which uses the conditional probabilities for words to occur in texts in which an option has been chosen to predict the probability of this option to be chosen in a new text for which only the words are known. Aeglos may thereby be trained directly from the data output of Angrist and is used in all question types to generate default values when it is activated.

The Documentation of this module is currently under preparation and will be available in autumn 2014.

7. Glossary

Angrist uses a small number of predefined functions which may be used in the project definition. Some of these functions have been mentioned or explained using the example project above. In this glossary, all functions are presented for a general overview. The functions are grouped and sorted by type and usage.

7.1. Initialization

`self.fuellen()`

This function initiates the GUI and sets the basic settings for each text. This includes the coder specific settings (by referring to `a_settings.ini`), the codebook (by referring to `a_codebook.ini`), the ID of the text (by referring to the todo-list) and manually defined settings which are used in the content analysis.

When this function is called, the content analysis of the current text is set back to zero. All entries are deleted and the content analysis starts from the initial page. Therefore, it is important not to call this function in the program without cautioning the coder and saving all data.

For the correct usage of `self.fuellen()`, refer to chapter 5.3.

`self.set_window()`

This function is called at the beginning of each session to set up the layout of the GUI. Any changes made in this function will affect the layout of all pages in the questionnaire.

For a detailed description of the layout, see chapter 2.

7.2. Project specific functions

`self.ask()`

This function is used to define the pages which are shown to the coder. The function contains a large `if..elif..else`-block in which a series of commands is defined for each value of `prog_pos`. Each block of commands usually contains a `self.buttons()`-command to define the buttons displayed on the page and a series of `self.question_XX()`-commands to call the appropriate question type for each variable.

For a detailed description of this function, refer to chapter 5.4.

`self.submit(overspill=0)`

This function is called whenever the coder hits the Check-button and thus submits the decisions to the program. Just like `self.ask()`, the Submit-Function contains a large `if..elif..else`-block defining the actions to be taken for each value of `prog_pos`.

The actions usually include storing the values to the internal storage, cleaning up the page, setting a new value for `prog_pos`, and calling the Ask-function again.

The Submit-function may take one argument. This is necessary for binding the function to events in tkinter. The argument may also be used to submit additional information to the function such as the identity of the button pressed.

For a detailed description of this function, refer to chapter 5.5.

`self.abort()`

This function is called by clicks on the Abort-button or by answers to buttons questions, which are linked to this function. Again, this function is defined for each value of `prog_pos` by means of an `if..elif..else`-block.

For a detailed description of this function, refer to chapter 5.6.

`self.back()`

This function is called by clicks on the Back-button. Its general setup sets back the program position by one page and calls the Ask-function again. If any other action is necessary to go back from a certain positon within the program, this action has to be defined using the `if..elif..else`-block.

For a detailed description of this function, refer to chapter 5.7.

`self.rb_tamper()`

This function is called by changing the value of any radiobutton question in the questionnaire. The actions to be taken when the value of a question is changed have to be defined for the respective page by means of an `if..elif..else`-block.

7.3. Important aides

This set of functions may be called in the Ask- or Submit-function to change the appearance of the GUI or to manipulate the data. The definition of these functions should only be tampered with by people with good knowledge of Python.

`self.show_review(level,rm=1,edit=0,height=3)`

This function displays a list of all items on a given level below the current level of analysis. This review list is displayed above the current page and there is a possibility to change the entries either by removing them from storage or editing them.

All entries in this list have the form '<*ID*>: *Trivialname*' where *ID* is the identifier of the unit of analysis for this entry and *Trivialname* is the value of the key `'#TN'` of this unit of analysis. The values for `'#TN'` are set automatically when using the `level_down()` function. They may then be changed manually if desired.

Options:

`level`	Either a string variable denominating the level of analysis or a list of strings denominating multiple levels of analysis. The string variables must be identical with the name of the level of analysis within the `storage` dictionary.
`rm`	Integer variable which may take the values 1 or 0. If set to 1, a remove-button is displayed next to the list which allows the coder to remove a unit of analysis from storage. Default: 1
`edit`	Integer variable which may take the values 1 or 0. If set to 1, a edit-button is displayed next to the list which allows the coder to edit a unit of analysis from storage. Default: 0 In order for this button to work, the function `self.edit_item()` has to be defined for the current project.
`height`	Height of the listbox in lines. Default: 3

Example:

```
self.review(['Actor','Issue'],1,0)
```

This example displays a list of all units of analysis that have been coded on the levels `'Actor'` and `'Issue'` which are below the current level of analysis. The coder may remove any item from the list but is not allowed to edit them.

`self.hide_review()`

This function removes the review list from the top of the current page. By default, this function is called for all pages defined in self.ask(). If a review list is desired it has to be called for each page where it is needed.

`self.remove_item(level, height)`

This function is called by pressing the remove button next to the review list. The parameters `level` and `height` are taken from the `self.show_review()` function and are used to display the list after removing an item. The function removes the selected item from storage and redisplays the list.

`self.edit_item(level)`

This function is called by pressing the edit button next to the review list. The parameter `level` is taken from the `self.show_review()` function. The function sets the data position to the selected unit of analysis and the program position to a page specified in the project. For each project this function has to be customized for the button to redirect the coder to the correct page within the questionnaire.

`self.level_up()`

This function changes the level of analysis to the parent level.

`self.level_down(variable,level)`

This function changes the level of analysis to a lower level. For this purpose, the values entered for a variable are used. This function is usually called in the Submit-function in order to change the unit of analysis according to coder inputs.

The function returns an integer with value 1 if the new unit of analysis was created successfully.

Options:

`variable`	String variable denominating the variable which is to be used to name the unit of analysis. The code of the value entered is used as internal identifier of the unit of analysis. The label of the option is used as trivialname and is stored in curr()['#TN'] for usage in other functions.
`level`	String variable denominating the level on which the unit of analysis is to be created.

Example:

When on text level and choosing actor 'Barrack Obama' with code 'us01' in variable 'Actor_ID' as actor to be coded in a new unit of analysis, the command:

```
self.level_down('Actor_ID','Actor')
```

Changes the value of `dta_pos` to `['Actor','us01']` and sets the value of `storage['Actor']['us01']['#TN']` to `'Barrack Obama'`. Accordingly, all future entries will be stored in this unit of analysis until the command `self.level_up()` sets the level of analysis back to text level.

`self.buttons(check=1,abort=0,back=1,pause=1)`

This function displays buttons below the current page of the questionnaire. If the function is called without any arguments, all buttons except for the Abort-Button are displayed. See also: chapter 2.

Options:

`check`	Integer variable with value 0 or 1. If set to 1, the Check-button is displayed. Default: 1
`abort`	Integer variable with value 0 or 1. If set to 1, the Abort-button is displayed. Default: 0
`back`	Integer variable with value 0 or 1. If set to 1, the Back-button is displayed. Default: 1
`pause`	Integer variable with value 0 or 1. If set to 1, the Break-button is displayed. Default: 1

Examples:

```
self.buttons(1,1,0,0)
```

Displays the Check- and Abort-buttons but suppresses Break and Back.

```
self.buttons(abort=1)
```

Displays all four buttons.

`self.check_entries()`

This function checks all answers for validity. The function returns `1` if all entries are valid and `0` if they are not. This function is called whenever the Submit-function is called. All commands in the Submit-function are only executed if `self.check_entries()` returns `1`.

The function returns 0 if:

- A dropdown or radiobutton question is answered with an option that has code '98'
- Text has been entered to a textbox which may not be processed

The function returns 1 if no error occurred or if `settings['Debugging']` is set to `'1'`.

`self.clean_up(pos, typ='')`

This function removes a question of type typ from position pos. If no value for parameter typ is provided, the function removes any question at this position. If there is no question with type typ at the given position, no action is taken.

Options:

`pos`	Integer variable with value 1, 2, or 3. Denominates the position of the question to remove.
`typ`	String variable naming the type of question to remove. Possible values: `'dd'`: dropdown question `'txt'`/`'txt2'`: text question `'rb'`: radiobutton question `'list'`: listbox `'listseek'`: searchable list `'listadd'`: list selection `'unit_auswahl'`: unit selection

`'rb'`/`'sd'`/`'rating'`/`'cb'`:	radiobutton, sd, rating, or checbox question
`'bt'`:	buttons question

`self.clean_up_all()`

This function cleans up the whole page, removing all questions by calling `self.clean_up()` for all positions.

`self.codegetter(variabel, item)`

This function looks up the code for an option for which only the name is known. It returns the code as a string variable.

Options:

`variable`	String variable denominating the variable which has the demanded option.
`item`	String variable with the exact label of the option which is demanded.

`self.namegetter(variable, item)`

This function looks up the label for an option for which only the code is known. It returns the label as a string variable.

Options:

`variable`	String variable denominating the variable which has the demanded option.
`item`	String variable with the exact code of the option which is demanded.

`self.unit_select(level):`

This function may be called to read the input of a `question_sel_units()` question and automatically open a new unit of analysis on the level specified. In addition, the function highlights the current unit of analysis in the text for better navigation.

The trivialname (`curr()['#TN']`) of this unit of analysis is set to the wording of the highlighted text.

Options:

`level`	String variable denominating the level on which the unit of analysis is to be created.

`self.unit_confirm(level,sel_tags=[]):`

This function is used to create a list of units of analysis from highlighted portions of the text. The function reads all marks in the text and translates them to potential units of analysis which may be displayed in a `question_sel_units()` question.

Options:

`level`	String variable denominating the level on which the unit of analysis is to be created.

`sel_tags` List of strings denominating the highlight-buttons which are to be read. If sel_tags is set to an empty list, all tags are converted to units of analysis. Default: `[]`

`self.store_var_all(setdef=0)`

This function stores all entries from the current page to the storage dictionary. All entries are stored at the position specified by dta_pos.

Options:

`setdef` Integer variable which may take the values 1 or 0. If set to 1, the current entries are set as default for future questions of the same variable. Default: `0`

`self.store_var(variable,pos=-1,setdef=0,store=1)`

This function stores the entry for a given variable to the storage dictionary. The function is usually called by self.store_var_all() but may also be used seperately in the Submit-function. The function returns the stored value.

Options:

`variable` String variable denominating the variable which is to be stored.

`pos` Position of the question on the current page.

`setdef` Integer variable which may take the values 1 or 0. If set to 1, the current entries are set as default for future questions of the same variable. Default: `0`

`store` Integer variable which may take the values 1 or 0. If set to 0, the value is not stored in `storage`. The function nevertheless returns the value. Default: `1`

The values returned and stored by this function depend on the type of question which was used to ask for the value:

- For dropdown and radiobutton questions, storage contains a tuple with two elements (Label,Code) as string variables as value.
- For text questions, storage contains a string variable as value.
- For checkbox, rating, and sd questions, storage contains a dictionary with all options as keys and the entered values as values. For rating scales, the values may be defined manually. For checkbox and sd questions, the values are integers.
- For List questions, storage contains a tuple of lists ([Label1,Label2...],[Code1, Code2..]) which contains all labels and codes selected. If no multiple selection is allowed, the lists only contain one element.

`self.message(m_id,m_type=1,variable='Err_Msg')`

This function displays a message of a given type to the coder. The message is displayed as a pop-up and may be closed by pressing 'OK' or closing the window.

Options:

`m_id` String variable denominating the code of the message to be shown. The code has to be defined in the codebook for the `variable` specified.

`m_type` Integer which may take the values 1, 2, or 3 and specifies the type of message to be

displayed.
`1`: Warning to the coder.
`2`: Information to the coder.
`3`: OK/Cancel-question. If this type of message is used, the function returns 1 if the coder pressed 'OK' and 0 if the coder pressed 'Cancel'. You may use this type of message for critical decisions.

`variable` String variable denominating the codebook variable in which the message is defined as an option. Default: `'Err_Msg'`

In the codebook, a set of error messages is defined:

```
[Err_Msg]
-
-
-
Caution01:Warning!#If you go back one step further, the coding will terminate
      and the text will be loaded again. All entries you made will then be lost
      and the coding process starts fresh for this text.##Are you sure you want
      to go back?
Caution02:This action will remove the text from the sample. Only discard texts
      that do not have anything to do with elections, immigration, or labor
      market policies.##Are you sure you want to discard this text?
Info01:Your insecurity has been transmitted. You may now continue coding.
Invalid-Selection01:No Item was selected from the list. Please select at least
      one item to continue.
Invalid-Selection02:Invalid selection. Please select an option to
      continue.##Variable:
Invalid-Selection03:Invalid Characters entered. Please do not use special
      characters.
Runtime-Error01:This action is impossible.##Please contact Martin Wettstein to
      find out why.
Runtime-Error02:There are no more items in the Todo-List or the Todo-List does
      not exist. Please contact the operational staff to get new texts or fix
      this problem.##If you do know the ID of your next text you may enter it
      manually.
Runtime-Error03:Please enter the text identifier. You may not continue coding
      without a valid identifier.
Runtime-Error04:The specified text was not found in the text-folder. Please
      check spelling.
Runtime-Error05:Please select Source and proceed to the next page before using
      this feature.##The highlighting of relevant words will depend on the
      country of origin of your Medium
```

You may define any other message in this variable or create new variables with messages to be called.

7.4. Question functions

Question functions are usually called in the Ask-function and are used to prompt data input by the coder. There is a variety of different question types available.

`self.question_dd(var,pos,width=40)`

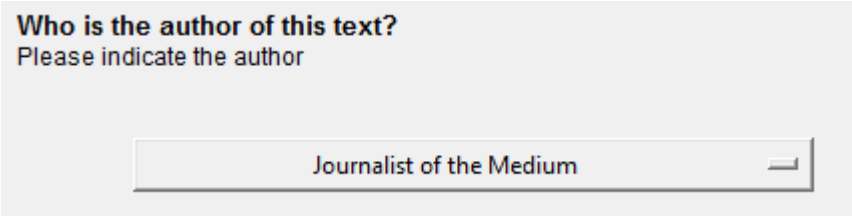
This function creates a dropdown question for variable `var` at position `pos`.

Options:

`var` String variable denominating the codebook variable to be asked.

`pos` Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.

`width`	Integer variable defining the width of the dropdown menu in characters. Default: `40`

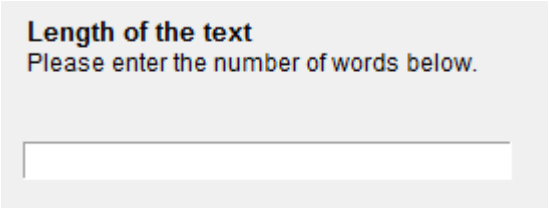


`self.question_txt(var,pos,width=40)`

This function creates a textbox question for variable `var` at position `pos`.

Options:

`var`	String variable denominating the codebook variable to be asked.
`pos`	Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.
`width`	Integer variable defining the width of the textbox in characters. Default: `40`

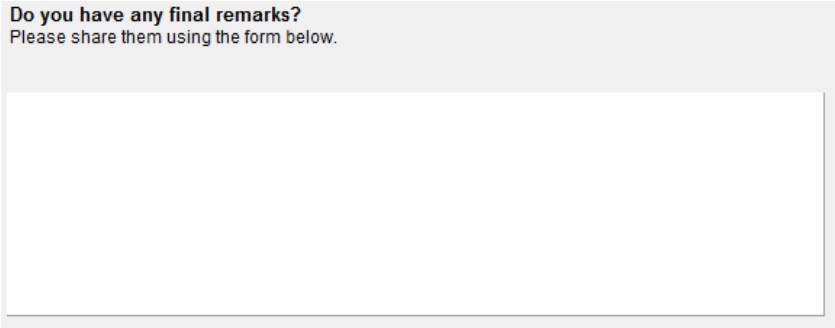


`self.question_txt2(var,pos,width=40,height=3,getselect=0)`

This function creates a textbox question with multiple lines for variable `var` at position `pos`.

Options:

`var`	String variable denominating the codebook variable to be asked.
`pos`	Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.
`width`	Integer variable defining the width of the textbox in characters. Default: `40`
`height`	Integer variable defining the height of the textbox in lines. Default: `3`
`getselect`	Integer variable which may be 1 or 0. If set to 1, a button is created to copy any text selection within the text display to the question. This saves the coder a Ctrl+C/Ctrl+V. Default: `0`



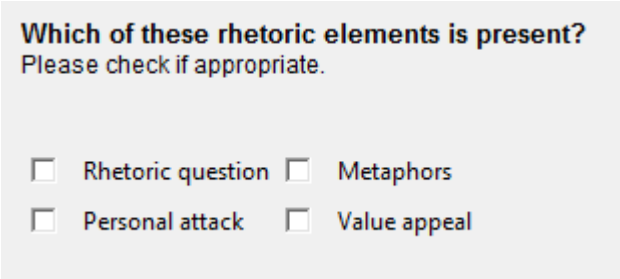
Example: self.question_txt2('Remarks',3,width=80,height=10)

`self.question_cb(var,pos,layout='hor',defval=0)`

This function creates a checkbox question for variable `var` at position `pos`. The question takes up to 14 options to be checked.

Options:

`var`	String variable denominating the codebook variable to be asked.
`pos`	Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.
`layout`	String variable which may take the values `'hor'` or `'vert'`. If set to `'hor'`, the options are presented in subsequent rows of 2 items each. If set to `'vert'`, the options are presented in subsequent columns of 7 items each. Default: `'hor'`
`defval`	Default value for all checkboxes. If set to 1 all checkboxes are selected. If set to 0, none is selected. This default value is overruled by any previous coding or explicitly set default value in the `def_val` directory. Default: `0`



`self.question_rb(var,pos,layout='vert',defval='98')`

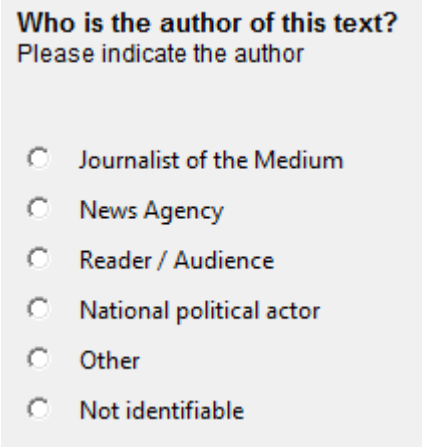
This function creates a radiobutton question for variable `var` at position `pos`. The question takes up to 14 options to be selected.

Options:

`var`	String variable denominating the codebook variable to be asked.
`pos`	Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.
`layout`	String variable which may take the values `'hor'` or `'vert'`. If set to `'hor'`, the options are presented in subsequent rows of 2 items each. If set to `'vert'`, the options are

presented in subsequent columns of 7 items each. Default: `'vert'`

`defval` Default value for the selection. This default value is overruled by any previous coding or explicitly set default value in the `def_val` directory. Default: `'98'`



`self.question_sd(var,pos,points=5,defval=0)`

This function creates a semantic differential question for variable `var` at position `pos`. The question takes up to 7 options to be rated on a scale between two extremes. The extreme values are the code and the label of each option of this variable. The code thereby represents the lowest rating (i.e. `0`), the label represents the highest rating.

Options:

`var` String variable denominating the codebook variable to be asked.

`pos` Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.

`points` Integer variable defining the number of points which may be selected between the extremes. Default: 5

`defval` Default value for all ratings. This default value is overruled by any previous coding or explicitly set default value in the `def_val` directory. Default: `0`

`self.question_rating(var, pos, scalelist=['disagree','','','','agree'], valuelist=['1','2','3','4','5'], defval='1')`

This function creates a rating question for variable `var` at position `pos`. The question takes up to 7 options to be rated on a scale which may be defined freely.

Options:

`var` String variable denominating the codebook variable to be asked.

`pos` Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.

`scalelist` List of strings which contains the labels of the rating scale. If a point does not have a

label, insert an empty string. Default: `['disagree','','','','agree']`

`valuelist` List of strings or integers which contains the values for each point which may be selected. The lenght of this list has to be identical to scalelist. Default: `['1','2','3','4','5']`

`defval` Default value for all ratings. This default value is overruled by any previous coding or explicitly set default value in the `def_val` directory. If `defval` is off the scale (i.e. not part of `valuelist`), no default value is set. Default: `'1'`

`self.question_bt(var,pos)`

This function creates a buttons question for variable `var` at position `pos`. The question takes up to four options to be displayed as buttons.

Options:

`var` String variable denominating the codebook variable to be asked.

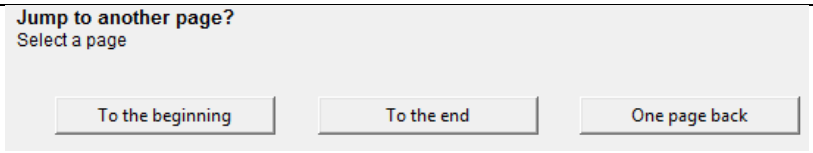
`pos` Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.

When a button is pressed, the Submit Function is called with the parameter 'button' indicating the button which was pressed. You may use this information within the Submit-Function to decide upon further actions. Just use the variable `button` which is set to the value of the option which was pressed.

Example:

This short example demonstrates the use of three buttons to jump forward and back in the questionnaire. This function is unlikely to be of any use but it shows the handling of this question type.

Codebook	`[Jump]`
	`Select a page`
	`Jump to another page?`
	`No Helptext available`
	`start:To the beginning`
	`end:To the end`
	`back:One page back`

Ask-Function	`elif prog_pos == 'jumping':` `  self.question_bt('Jump',1)`
Visualization	

Submit-Function	`elif prog_pos == 'jumping':`
	`  If button == 'start':`
	`  prog_pos = 'firstpage'`
	`  elif button == 'end':`
	`  prog_pos = 'last_page'`
	`  elif button == 'back':`
	`  prog_pos = settings['Page_History'][-2]`
	`  self.ask()`

`self.question_ls(var,liste,multi=1)`

This function creates a listbox question for variable `var` at position `pos`.

Options:

`var`	String variable denominating the codebook variable to be asked.
`liste`	String variable denominating the codebook variable which contains the list. This may be the same variable as `var` but does not have to be.
`multi`	Integer which may take the values 1 or 0. If set to 1, multiple selection of items is possible.

`self.question_lseek(var,liste,multi=0)`

This function creates a searchable listbox question for variable `var` at position `pos`.

Options:

`var`	String variable denominating the codebook variable to be asked.
`liste`	String variable denominating the codebook variable which contains the list. This may be the same variable as `var` but does not have to be.
`multi`	Integer which may take the values 1 or 0. If set to 1, multiple selection of items is possible.

To search the list the coder may input any string of characters to a textbox above the list. The list will then automatically be reduced to present only the items with this sting inside. The search term is not case-sensitive and does not have to be in full words.

`self.question_ladd(var,liste,multi=0)`

This function creates a searchable listbox question for variable `var` at position `pos` from which single items may be added to an answer list.

Options:

`var`	String variable denominating the codebook variable to be asked.
`liste`	String variable denominating the codebook variable which contains the list. This may be the same variable as `var` but does not have to be.
`multi`	Integer which may take the values 1 or 0. If set to 1, multiple selection of items is possible.

To search the list the coder may input any string of characters to a textbox above the list. The list will then automatically be reduced to present only the items with this sting inside. The search term is not case-sensitive and does not have to be in full words.

To add an item to the selections list or to remove an item, the coder may use the buttons 'Add Item' and 'Remove Item' which are displayed between the two lists.

`self.question_mark_units(var,level)`

This function asks the coder to mark all units of analysis on a given `level`. The exact question is taken from variable `var`.

Options:

`var`	String variable denominating the codebook variable to be asked. This variable has exactly one option which has the label 'Done', 'Finished', or any other confirmation. If there is no such option, it is recommended to leave the Check-button on display.
`level`	String variable denominating the level of analysis on which the units are to be coded.

`self.question_sel_units(var,level)`

This function asks the coder to select a unit of analysis from a given `level`. The exact question is taken from variable `var`. The list of units of analysis is created from text highlights previously stored using the function `self.unit_confirm()`.

Options:

`var`	String variable denominating the codebook variable to be asked.
`level`	String variable denominating the level of analysis on which the units are to be coded.

`self.question_menu(var,position)`

This function creates a menu question for variable `var` at position `pos`. The question takes a hierarchically ordered set of options which may be used similar to the button question. Each option calls the submit function with the parameter `button` set to the code of the selected option.

Options:

`var`	String variable denominating the codebook variable to be asked.
`pos`	Integer variable which may take the values 1, 2, or 3 and which indicates the position of the question on the page.

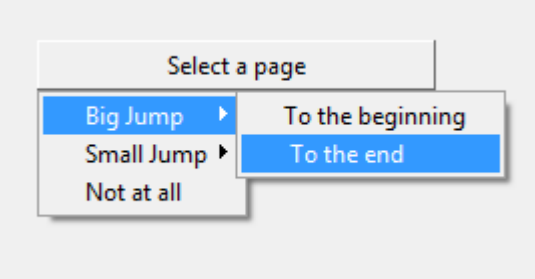
You may use the variable `button` within the Submit-Function to decide upon further actions. The values of this variable correspond to the code of the option selected.

Example:

This short example demonstrates the use of a small menu to jump forward and back in the questionnaire. This function is unlikely to be of any use but it shows the handling of this question type.

Codebook	`[Jump]`
	`Select a page`
	`Jump to another page?`
	`No Helptext available`
	`1:Big Jump`
	`-start:To the beginning`
	`-end: To the end`
	`2:Small Jump`
	`-back:One page back`
	`none:Not at all`

Ask-Function	`elif prog_pos == 'jumping':` `  self.question_menu('Jump',1)`

Visualization	

Submit-Function	`elif prog_pos == 'jumping':`
	`  If button == 'start':`
	`  prog_pos = 'firstpage'`
	`  elif button == 'end':`
	`  prog_pos = 'last_page'`
	`  elif button == 'back':`
	`  prog_pos = settings['Page_History'][-2]`
	`  self.ask()`

7.5. Basic functions

The basic functions are only of interest for developers and should not be tampered with. These functions may be called in some situations; most of them are only called by other functions and are invisible during project development. Nevertheless, the most important ones are briefly introduced here.

`baum_schreiben(direc)`

This function transforms a variable of the type directory to a string variable containing an indented tree representation. `direc` is the name of the directory variable to be transformed.

`bereinigen(uml_string,lc=0,lb=0,uml=0)`

This function takes a string with special characters and transforms it to ASCII. There are some useful options there:

Options:

`uml_string`	String variable or unicode string with special characters.
`lc`	If set to 1, the output is completely in lower case
`lb`	If set to 1, the output contains line-breaks. If set to 0, the line-breaks are replaced by the string `' / '`.
`uml`	If set to 1, the output contains German Umlauts. They may be used in the display of a text but not in the storage thereof. For the purpose of storage, `uml` has to be set to 0.

`self.clean_all_tags(sel_tag=[]):`

This function cleans all highlights from the text display. If desired, only certain types of highlights may be removed by specifying the parameter `sel_tag`.

`self.ausblenden()`

This function removes the text display and all highlight buttons from the GUI.

7.6. Input and output of data

Finally, there is a set of functions which may be used to import and export data in Angrist. These functions may be used to load texts, save data, or import and export tables.

`self.export_data(dta_pos_all, varlist, filename, debug=0)`

This function exports data from the storage directory to rectangular data matrices. The matrices are created as text-files with tabulators as cell delimiters. Each row represents a case. For each level of analysis, an individual may be generated using this function.

Options:

`dta_pos_all`	List of strings containing the level of analysis to be exported and all parent levels of analysis. The order is descending by depth of levels of analysis. Example: If level `'Statement'` should be exported which is below the level of `'Actor'` which is below the level of the text, the value of `dta_pos_all` is: `['Actor','Statement']`. Put differently, the list here represents the odd entries in `dta_pos`.
`varlist`	List of strings denominating all variables to be exported in this table. If any variable name does not exist in the data an empty cell will be exported.
`filename`	A string denominating the file in which the data is to be exported. If the file does not exist yet, a new file will be created with variable names in the top row.
`debug`	If set to 1, all cells will not only contain the codes but also the name of the variable this code belongs to. This may be necessary to debug the export function.

`baum_export()`

This function exports the complete contents of the dictionary `storage` to a file specified in the `settings` dictionary. If there is a valid entry for `settings['Out_Tree']`, an indented representation of the dictionary is written to the file specified. If there is a valid entry for `settings['Out_JSON']`, a JSON-formated representation is written to a file.

`artikelholen(ID)`

This function reads a textfile with name ID to be displayed in the text display. The function always calls the function `textmine()` to which it submits a list of all lines in the text file.

`textmine(linelist)`

This function may be used to set default values of variables on the base of patterns in the text. You may, for example, extract informations from the header of a text, count words, or look for certain actors. This function is called whenever a new text is loaded to the text display.

`get_codebook(filename)`

This function loads a codebook using the format of Angrist codebooks to a dictionary of lists. See chapter 4.3 for detailed information.

`get_todo(filename)`

This function reads a list of text IDs to return the topmost entry. This function is used to get the next item on the todo list.

`check_todo(filename)`

This function changes a todo list in a given file by removing the current text ID from the list. This function may be called at the end of data collection to update the todo list.

`get_data(filename, varlist=[])`

This function reads a data matrix stored in a text file with tabulators as cell delimiters. The data is stored in a dictionary of lists which each contain all cells in a column. The key for each list is the topmost entry (usually the name of the column).

Options:

`varlist`	List of variables to be loaded. If omitted, the first entry is used as column name.

`get_varnames(filename)`

This function extracts the header of a data matrix stored in a text file with tabulators as cell delimiters. This header usually contains the names of the columns.

`write_data(data, varlist, filename)`

This function writes a data dictionary (as produced by `get_data()`) to a text-file with tabulators as cell delimiters.

8. Impressum

Programmer:

Martin Wettstein

Institut für Publizistikwissenschaften und

Medienforschung der Universität Zürich (IPMZ)

Andreasstrasse 15

8050 Zürich

m.wettstein@ipmz.uzh.ch

Tel: 044 635 20 78